



ESP32

ESP-IDF Programming Guide



Release v4.1.3
Espressif Systems
Apr 13, 2022




Table of contents

| | |
|--|------------|
| Table of contents | i |
| 1 Get Started | 3 |
| 1.1 Introduction | 3 |
| 1.2 What You Need | 3 |
| 1.3 Development Board Overviews | 4 |
| 1.3.1 ESP32-DevKitC V4 Getting Started Guide | 4 |
| 1.3.2 ESP-WROVER-KIT V4.1 Getting Started Guide | 9 |
| 1.3.3 ESP32-PICO-KIT V4 / V4.1 Getting Started Guide | 31 |
| 1.3.4 ESP32-Ethernet-Kit V1.1 Getting Started Guide | 41 |
| 1.4 Installation Step by Step | 58 |
| 1.4.1 Setting up Development Environment | 58 |
| 1.4.2 Creating Your First Project | 58 |
| 1.5 Step 1. Install prerequisites | 58 |
| 1.5.1 Standard Setup of Toolchain for Windows | 59 |
| 1.5.2 Standard Setup of Toolchain for Linux | 64 |
| 1.5.3 Standard Setup of Toolchain for Mac OS | 64 |
| 1.6 Step 2. Get ESP-IDF | 66 |
| 1.6.1 Linux and macOS | 66 |
| 1.6.2 Windows | 66 |
| 1.7 Step 3. Set up the tools | 66 |
| 1.7.1 Windows | 67 |
| 1.7.2 Linux and macOS | 67 |
| 1.7.3 Alternative File Downloads | 67 |
| 1.7.4 Customizing the tools installation path | 68 |
| 1.8 Step 4. Set up the environment variables | 68 |
| 1.8.1 Windows | 68 |
| 1.8.2 Linux and macOS | 68 |
| 1.9 Step 5. Start a Project | 68 |
| 1.9.1 Linux and macOS | 69 |
| 1.9.2 Windows | 69 |
| 1.10 Step 6. Connect Your Device | 69 |
| 1.11 Step 7. Configure | 69 |
| 1.11.1 Linux and macOS | 69 |
| 1.11.2 Windows | 69 |
| 1.12 Step 8. Build the Project | 70 |
| 1.13 Step 9. Flash onto the Device | 71 |
| 1.14 Step 10. Monitor | 72 |
| 1.15 Updating ESP-IDF | 73 |
| 1.16 Related Documents | 73 |
| 1.16.1 Establish Serial Connection with ESP32 | 73 |
| 1.16.2 Build and Flash with Eclipse IDE | 79 |
| 1.16.3 IDF Monitor | 79 |
| 1.16.4 Customized Setup of Toolchain | 83 |
| 1.16.5 Get Started (Legacy GNU Make) | 88 |
| 2 API Reference | 113 |

| | | |
|--------|---------------------------------|-----|
| 2.1 | Bluetooth API | 113 |
| 2.1.1 | Controller && VHCI | 113 |
| 2.1.2 | BT COMMON | 120 |
| 2.1.3 | BT LE | 124 |
| 2.1.4 | CLASSIC BT | 200 |
| 2.1.5 | NimBLE-based host APIs | 275 |
| 2.1.6 | ESP-BLE-MESH | 277 |
| 2.2 | Networking APIs | 489 |
| 2.2.1 | Wi-Fi | 489 |
| 2.2.2 | Ethernet | 559 |
| 2.2.3 | IP Network Layer | 573 |
| 2.2.4 | Application Layer | 588 |
| 2.3 | Peripherals API | 588 |
| 2.3.1 | Analog to Digital Converter | 588 |
| 2.3.2 | Controller Area Network (CAN) | 602 |
| 2.3.3 | Digital To Analog Converter | 617 |
| 2.3.4 | GPIO & RTC GPIO | 620 |
| 2.3.5 | I2C Driver | 633 |
| 2.3.6 | I2S | 646 |
| 2.3.7 | LED Control | 657 |
| 2.3.8 | MCPWM | 669 |
| 2.3.9 | Pulse Counter | 687 |
| 2.3.10 | RMT | 695 |
| 2.3.11 | SD Pull-up Requirements | 711 |
| 2.3.12 | SDMMC Host Driver | 715 |
| 2.3.13 | SD SPI Host Driver | 719 |
| 2.3.14 | SDIO Card Slave Driver | 722 |
| 2.3.15 | Sigma-delta Modulation | 732 |
| 2.3.16 | SPI Master Driver | 734 |
| 2.3.17 | SPI Slave Driver | 752 |
| 2.3.18 | ESP32-S2 Temperature Sensor | 759 |
| 2.3.19 | Timer | 761 |
| 2.3.20 | Touch Sensor | 771 |
| 2.3.21 | UART | 777 |
| 2.4 | Application Protocols | 799 |
| 2.4.1 | ASIO port | 799 |
| 2.4.2 | ESP-MQTT | 800 |
| 2.4.3 | ESP-TLS | 811 |
| 2.4.4 | ESP HTTP Client | 820 |
| 2.4.5 | HTTP Server | 832 |
| 2.4.6 | HTTPS server | 852 |
| 2.4.7 | ICMP Echo | 854 |
| 2.4.8 | ESP Local Control | 859 |
| 2.4.9 | mDNS Service | 866 |
| 2.4.10 | ESP-Modbus | 874 |
| 2.4.11 | ESP WebSocket Client | 879 |
| 2.4.12 | ESP Serial Slave Link | 884 |
| 2.4.13 | IP Network Layer | 889 |
| 2.5 | Provisioning API | 890 |
| 2.5.1 | Protocol Communication | 890 |
| 2.5.2 | Unified Provisioning | 899 |
| 2.5.3 | Wi-Fi Provisioning | 903 |
| 2.6 | Storage API | 919 |
| 2.6.1 | FAT Filesystem Support | 919 |
| 2.6.2 | Manufacturing Utility | 923 |
| 2.6.3 | Non-volatile storage library | 928 |
| 2.6.4 | NVS Partition Generator Utility | 946 |
| 2.6.5 | SD/SDIO/MMC Driver | 953 |

| | | |
|----------|---|-------------|
| 2.6.6 | SPI Flash API | 962 |
| 2.6.7 | SPIFFS Filesystem | 981 |
| 2.6.8 | Virtual filesystem component | 985 |
| 2.6.9 | Wear Levelling API | 994 |
| 2.7 | System API | 998 |
| 2.7.1 | App Image Format | 998 |
| 2.7.2 | Application Level Tracing | 1002 |
| 2.7.3 | eFuse Manager | 1007 |
| 2.7.4 | Error Codes and Helper Functions | 1017 |
| 2.7.5 | ESP HTTPS OTA | 1019 |
| 2.7.6 | ESP-pthread | 1022 |
| 2.7.7 | Event Loop Library | 1024 |
| 2.7.8 | FreeRTOS | 1037 |
| 2.7.9 | FreeRTOS Additions | 1115 |
| 2.7.10 | Heap Memory Allocation | 1131 |
| 2.7.11 | Heap Memory Debugging | 1142 |
| 2.7.12 | High Resolution Timer | 1152 |
| 2.7.13 | The himem allocation API | 1156 |
| 2.7.14 | Inter-Processor Call | 1158 |
| 2.7.15 | Call function with external stack | 1160 |
| 2.7.16 | Interrupt allocation | 1161 |
| 2.7.17 | Logging library | 1166 |
| 2.7.18 | Miscellaneous System APIs | 1171 |
| 2.7.19 | Over The Air Updates (OTA) | 1178 |
| 2.7.20 | Performance Monitor | 1187 |
| 2.7.21 | Power Management | 1190 |
| 2.7.22 | Sleep Modes | 1195 |
| 2.7.23 | Watchdogs | 1203 |
| 2.7.24 | System Time | 1206 |
| 2.8 | Project Configuration | 1210 |
| 2.8.1 | Introduction | 1210 |
| 2.8.2 | Project Configuration Menu | 1210 |
| 2.8.3 | Using sdkconfig.defaults | 1211 |
| 2.8.4 | Kconfig Formatting Rules | 1211 |
| 2.8.5 | Backward Compatibility of Kconfig Options | 1211 |
| 2.8.6 | Configuration Options Reference | 1212 |
| 2.8.7 | Customisations | 1377 |
| 2.9 | Error Codes Reference | 1377 |
| 3 | ESP32 Hardware Reference | 1383 |
| 3.1 | ESP32 Modules and Boards | 1383 |
| 3.1.1 | Modules | 1383 |
| 3.1.2 | Development Boards | 1386 |
| 3.1.3 | Related Documents | 1389 |
| 3.2 | Previous Versions of ESP32 Modules and Boards | 1389 |
| 3.2.1 | Modules (No updated or discontinued modules) | 1389 |
| 3.2.2 | Development Boards | 1389 |
| 3.2.3 | Related Documents | 1394 |
| 4 | ESP32-S2 Beta Preview Support | 1395 |
| 4.1 | Documentation | 1395 |
| 4.2 | Setting up for ESP32-S2 Beta | 1395 |
| 4.2.1 | Manual toolchain URLs | 1395 |
| 4.3 | Building a project for ESP32-S2 Beta | 1396 |
| 4.4 | Limitations | 1396 |
| 4.5 | Roadmap | 1396 |
| 4.6 | Issues | 1397 |

| | | |
|----------|--|-------------|
| 5 | API Guides | 1399 |
| 5.1 | Application Level Tracing library | 1399 |
| 5.1.1 | Overview | 1399 |
| 5.1.2 | Modes of Operation | 1399 |
| 5.1.3 | Configuration Options and Dependencies | 1400 |
| 5.1.4 | How to use this library | 1401 |
| 5.2 | BluFi | 1410 |
| 5.2.1 | Overview | 1410 |
| 5.2.2 | The BluFi Flow | 1410 |
| 5.2.3 | The flow chart of BluFi | 1411 |
| 5.2.4 | The Frame Formats Defined in BluFi | 1411 |
| 5.2.5 | The Security Implementation of ESP32 | 1415 |
| 5.2.6 | GATT Related Instructions | 1416 |
| 5.3 | Bootloader | 1416 |
| 5.3.1 | Bootloader compatibility | 1416 |
| 5.3.2 | FACTORY reset | 1417 |
| 5.3.3 | Boot from TEST firmware | 1418 |
| 5.3.4 | Fast boot from Deep Sleep | 1418 |
| 5.3.5 | Customer bootloader | 1418 |
| 5.4 | Build System | 1418 |
| 5.4.1 | Overview | 1419 |
| 5.4.2 | Using the Build System | 1419 |
| 5.4.3 | Example Project | 1422 |
| 5.4.4 | Project CMakeLists File | 1423 |
| 5.4.5 | Component CMakeLists Files | 1424 |
| 5.4.6 | Component Configuration | 1426 |
| 5.4.7 | Preprocessor Definitions | 1427 |
| 5.4.8 | Component Requirements | 1427 |
| 5.4.9 | Overriding Parts of the Project | 1430 |
| 5.4.10 | Configuration-Only Components | 1430 |
| 5.4.11 | Debugging CMake | 1431 |
| 5.4.12 | Example Component CMakeLists | 1431 |
| 5.4.13 | Custom sdkconfig defaults | 1435 |
| 5.4.14 | Flash arguments | 1435 |
| 5.4.15 | Building the Bootloader | 1436 |
| 5.4.16 | Selecting the Target | 1436 |
| 5.4.17 | Writing Pure CMake Components | 1436 |
| 5.4.18 | Using Third-Party CMake Projects with Components | 1437 |
| 5.4.19 | Using Prebuilt Libraries with Components | 1438 |
| 5.4.20 | Using ESP-IDF in Custom CMake Projects | 1438 |
| 5.4.21 | ESP-IDF CMake Build System API | 1439 |
| 5.4.22 | File Globbing & Incremental Builds | 1442 |
| 5.4.23 | Build System Metadata | 1443 |
| 5.4.24 | Build System Internals | 1444 |
| 5.4.25 | Migrating from ESP-IDF GNU Make System | 1446 |
| 5.5 | Build System (Legacy GNU Make) | 1448 |
| 5.5.1 | Using the Build System | 1448 |
| 5.5.2 | Overview | 1448 |
| 5.5.3 | Building the Bootloader | 1458 |
| 5.6 | Console | 1458 |
| 5.6.1 | Line editing | 1459 |
| 5.6.2 | Splitting of command line into arguments | 1460 |
| 5.6.3 | Argument parsing | 1460 |
| 5.6.4 | Command registration and dispatching | 1460 |
| 5.6.5 | Example | 1461 |
| 5.7 | Deep Sleep Wake Stubs | 1461 |
| 5.7.1 | Rules for Wake Stubs | 1461 |
| 5.7.2 | Implementing A Stub | 1461 |

| | | |
|--------|--|------|
| 5.7.3 | Loading Code Into RTC Memory | 1462 |
| 5.7.4 | Loading Data Into RTC Memory | 1462 |
| 5.8 | Error Handling | 1463 |
| 5.8.1 | Overview | 1463 |
| 5.8.2 | Error codes | 1463 |
| 5.8.3 | Converting error codes to error messages | 1463 |
| 5.8.4 | ESP_ERROR_CHECK macro | 1464 |
| 5.8.5 | Error handling patterns | 1464 |
| 5.8.6 | C++ Exceptions | 1465 |
| 5.9 | ESP-BLE-MESH | 1465 |
| 5.9.1 | Getting Started with ESP-BLE-MESH | 1466 |
| 5.9.2 | ESP-BLE-MESH Examples | 1473 |
| 5.9.3 | ESP-BLE-MESH Demo Videos | 1473 |
| 5.9.4 | ESP-BLE-MESH FAQ | 1473 |
| 5.9.5 | Related Documents | 1473 |
| 5.10 | ESP-MESH | 1502 |
| 5.10.1 | Overview | 1502 |
| 5.10.2 | Introduction | 1502 |
| 5.10.3 | ESP-MESH Concepts | 1503 |
| 5.10.4 | Building a Network | 1509 |
| 5.10.5 | Managing a Network | 1514 |
| 5.10.6 | Data Transmission | 1517 |
| 5.10.7 | Channel Switching | 1519 |
| 5.10.8 | Performance | 1522 |
| 5.10.9 | Further Notes | 1523 |
| 5.11 | ESP32 Core Dump | 1523 |
| 5.11.1 | Overview | 1523 |
| 5.11.2 | Configuration | 1523 |
| 5.11.3 | Save core dump to flash | 1524 |
| 5.11.4 | Print core dump to UART | 1524 |
| 5.11.5 | ROM Functions in Backtraces | 1524 |
| 5.11.6 | Running ‘espcoredump.py’ | 1524 |
| 5.12 | Event Handling | 1525 |
| 5.12.1 | Wi-Fi, Ethernet, and IP Events | 1525 |
| 5.12.2 | Mesh Events | 1527 |
| 5.12.3 | Bluetooth Events | 1527 |
| 5.13 | Support for external RAM | 1527 |
| 5.13.1 | Introduction | 1527 |
| 5.13.2 | Hardware | 1527 |
| 5.13.3 | Configuring External RAM | 1528 |
| 5.13.4 | Restrictions | 1529 |
| 5.13.5 | Chip revisions | 1529 |
| 5.14 | Fatal Errors | 1530 |
| 5.14.1 | Overview | 1530 |
| 5.14.2 | Panic Handler | 1530 |
| 5.14.3 | Register Dump and Backtrace | 1531 |
| 5.14.4 | GDB Stub | 1533 |
| 5.14.5 | Guru Meditation Errors | 1534 |
| 5.14.6 | Other Fatal Errors | 1535 |
| 5.15 | Flash Encryption | 1536 |
| 5.15.1 | Introduction | 1536 |
| 5.15.2 | eFuse Used During Flash Encryption Process | 1537 |
| 5.15.3 | Flash Encryption Process | 1537 |
| 5.15.4 | Steps to Setup Flash Encryption | 1538 |
| 5.15.5 | Key Points About Flash Encryption | 1545 |
| 5.15.6 | Using Encrypted Flash | 1545 |
| 5.15.7 | Updating Encrypted Flash | 1546 |
| 5.15.8 | Disabling Flash Encryption | 1546 |


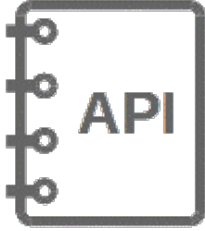

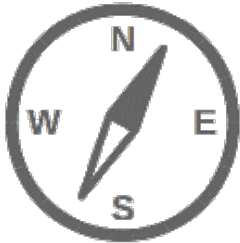
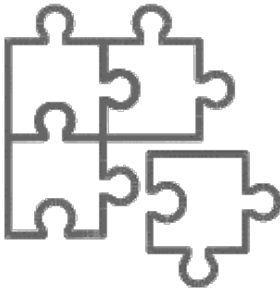

| | | |
|---------|--|------|
| 5.15.9 | Limitations of Flash Encryption | 1546 |
| 5.15.10 | Flash Encryption and Secure Boot | 1547 |
| 5.15.11 | Advanced Features | 1547 |
| 5.15.12 | Technical Details | 1548 |
| 5.16 | ESP-IDF FreeRTOS SMP Changes | 1549 |
| 5.16.1 | Overview | 1549 |
| 5.16.2 | Backported Features | 1550 |
| 5.16.3 | Tasks and Task Creation | 1551 |
| 5.16.4 | Scheduling | 1551 |
| 5.16.5 | Critical Sections & Disabling Interrupts | 1554 |
| 5.16.6 | Floating Point Arithmetic | 1554 |
| 5.16.7 | Task Deletion | 1555 |
| 5.16.8 | Thread Local Storage Pointers & Deletion Callbacks | 1555 |
| 5.16.9 | Configuring ESP-IDF FreeRTOS | 1555 |
| 5.17 | General Notes About ESP-IDF Programming | 1556 |
| 5.17.1 | Application startup flow | 1556 |
| 5.17.2 | Application memory layout | 1557 |
| 5.17.3 | DMA Capable Requirement | 1559 |
| 5.18 | High-Level Interrupts | 1560 |
| 5.18.1 | Interrupt Levels | 1560 |
| 5.18.2 | Notes | 1561 |
| 5.19 | JTAG Debugging | 1561 |
| 5.19.1 | Introduction | 1561 |
| 5.19.2 | How it Works? | 1562 |
| 5.19.3 | Selecting JTAG Adapter | 1562 |
| 5.19.4 | Setup of OpenOCD | 1563 |
| 5.19.5 | Configuring ESP32 Target | 1563 |
| 5.19.6 | Launching Debugger | 1568 |
| 5.19.7 | Debugging Examples | 1568 |
| 5.19.8 | Building OpenOCD from Sources | 1569 |
| 5.19.9 | Tips and Quirks | 1573 |
| 5.19.10 | Related Documents | 1573 |
| 5.20 | Linker Script Generation | 1601 |
| 5.20.1 | Overview | 1601 |
| 5.20.2 | Quick Start | 1602 |
| 5.20.3 | Linker Script Generation Internals | 1605 |
| 5.21 | lwIP | 1610 |
| 5.21.1 | Supported APIs | 1610 |
| 5.21.2 | BSD Sockets API | 1610 |
| 5.21.3 | Netconn API | 1614 |
| 5.21.4 | lwIP FreeRTOS Task | 1614 |
| 5.21.5 | esp-lwip custom modifications | 1615 |
| 5.21.6 | Performance Optimization | 1616 |
| 5.22 | Partition Tables | 1617 |
| 5.22.1 | Overview | 1617 |
| 5.22.2 | Built-in Partition Tables | 1617 |
| 5.22.3 | Creating Custom Tables | 1618 |
| 5.22.4 | Generating Binary Partition Table | 1619 |
| 5.22.5 | Flashing the partition table | 1620 |
| 5.22.6 | Partition Tool (parttool.py) | 1620 |
| 5.23 | RF calibration | 1622 |
| 5.23.1 | Partial calibration | 1622 |
| 5.23.2 | Full calibration | 1622 |
| 5.23.3 | No calibration | 1622 |
| 5.23.4 | PHY initialization data | 1622 |
| 5.24 | ESP32 ROM console | 1623 |
| 5.24.1 | Full list of supported statements and functions | 1623 |
| 5.24.2 | Example programs | 1624 |

| | | |
|---------|--|------|
| 5.24.3 | Credits | 1624 |
| 5.25 | Secure Boot | 1624 |
| 5.25.1 | Background | 1625 |
| 5.25.2 | Secure Boot Process Overview | 1625 |
| 5.25.3 | Keys | 1625 |
| 5.25.4 | Bootloader Size | 1626 |
| 5.25.5 | How To Enable Secure Boot | 1626 |
| 5.25.6 | Re-Flashable Software Bootloader | 1627 |
| 5.25.7 | Generating Secure Boot Signing Key | 1628 |
| 5.25.8 | Remote Signing of Images | 1628 |
| 5.25.9 | Secure Boot Best Practices | 1628 |
| 5.25.10 | Technical Details | 1628 |
| 5.25.11 | Secure Boot & Flash Encryption | 1630 |
| 5.25.12 | Signed App Verification Without Hardware Secure Boot | 1630 |
| 5.25.13 | Advanced Features | 1630 |
| 5.26 | Secure Boot V2 | 1631 |
| 5.26.1 | Background | 1631 |
| 5.26.2 | Advantages | 1631 |
| 5.26.3 | Secure Boot V2 Process | 1631 |
| 5.26.4 | Signature Block Format | 1632 |
| 5.26.5 | Verifying the signature Block | 1632 |
| 5.26.6 | Bootloader Size | 1632 |
| 5.26.7 | eFuse usage | 1633 |
| 5.26.8 | How To Enable Secure Boot V2 | 1633 |
| 5.26.9 | Restrictions after Secure Boot is enabled | 1634 |
| 5.26.10 | Generating Secure Boot Signing Key | 1634 |
| 5.26.11 | Remote Signing of Images | 1634 |
| 5.26.12 | Secure Boot Best Practices | 1635 |
| 5.26.13 | Technical Details | 1635 |
| 5.26.14 | Secure Boot & Flash Encryption | 1635 |
| 5.26.15 | Advanced Features | 1635 |
| 5.27 | Thread Local Storage | 1635 |
| 5.27.1 | Overview | 1635 |
| 5.27.2 | FreeRTOS Native API | 1636 |
| 5.27.3 | Pthread API | 1636 |
| 5.27.4 | C11 Standard | 1636 |
| 5.28 | Tools | 1636 |
| 5.28.1 | IDF Docker Image | 1636 |
| 5.28.2 | IDF Component Manager | 1638 |
| 5.29 | ULP Coprocessor programming | 1639 |
| 5.29.1 | ESP32 ULP coprocessor instruction set | 1639 |
| 5.29.2 | ESP32-S2 ULP coprocessor instruction set | 1652 |
| 5.29.3 | Programming ULP coprocessor using C macros | 1668 |
| 5.29.4 | Installing the Toolchain | 1673 |
| 5.29.5 | Compiling the ULP Code | 1673 |
| 5.29.6 | Accessing the ULP Program Variables | 1674 |
| 5.29.7 | Starting the ULP Program | 1675 |
| 5.29.8 | ULP Program Flow | 1676 |
| 5.30 | The ULP Coprocessor (Legacy GNU Make) | 1677 |
| 5.30.1 | Installing the Toolchain | 1677 |
| 5.30.2 | Compiling the ULP Code | 1677 |
| 5.30.3 | Accessing the ULP Program Variables | 1678 |
| 5.30.4 | Starting the ULP Program | 1679 |
| 5.30.5 | ULP Program Flow | 1680 |
| 5.31 | Unit Testing (Legacy GNU Make) | 1681 |
| 5.31.1 | Normal Test Cases | 1681 |
| 5.31.2 | Multi-device Test Cases | 1681 |
| 5.31.3 | Multi-stage Test Cases | 1682 |

| | | |
|----------|--|-------------|
| 5.31.4 | Building Unit Test App | 1683 |
| 5.31.5 | Running Unit Tests | 1683 |
| 5.32 | Unit Testing in ESP32 | 1684 |
| 5.32.1 | Normal Test Cases | 1685 |
| 5.32.2 | Multi-device Test Cases | 1685 |
| 5.32.3 | Multi-stage Test Cases | 1686 |
| 5.32.4 | Building Unit Test App | 1687 |
| 5.32.5 | Running Unit Tests | 1687 |
| 5.32.6 | Timing Code with Cache Compensated Timer | 1688 |
| 5.33 | Wi-Fi Driver | 1689 |
| 5.33.1 | ESP32 Wi-Fi Feature List | 1689 |
| 5.33.2 | How To Write a Wi-Fi Application | 1689 |
| 5.33.3 | ESP32 Wi-Fi API Error Code | 1690 |
| 5.33.4 | ESP32 Wi-Fi API Parameter Initialization | 1690 |
| 5.33.5 | ESP32 Wi-Fi Programming Model | 1690 |
| 5.33.6 | ESP32 Wi-Fi Event Description | 1691 |
| 5.33.7 | ESP32 Wi-Fi Station General Scenario | 1694 |
| 5.33.8 | ESP32 Wi-Fi AP General Scenario | 1696 |
| 5.33.9 | ESP32 Wi-Fi Scan | 1696 |
| 5.33.10 | ESP32 Wi-Fi Station Connecting Scenario | 1704 |
| 5.33.11 | ESP32 Wi-Fi Station Connecting When Multiple APs Are Found | 1708 |
| 5.33.12 | Wi-Fi Reconnect | 1708 |
| 5.33.13 | Wi-Fi Beacon Timeout | 1708 |
| 5.33.14 | ESP32 Wi-Fi Configuration | 1708 |
| 5.33.15 | Wi-Fi Security | 1714 |
| 5.33.16 | ESP32 Wi-Fi Power-saving Mode | 1715 |
| 5.33.17 | ESP32 Wi-Fi Connect Crypto | 1715 |
| 5.33.18 | ESP32 Wi-Fi Throughput | 1716 |
| 5.33.19 | Wi-Fi 80211 Packet Send | 1716 |
| 5.33.20 | Wi-Fi Sniffer Mode | 1718 |
| 5.33.21 | Wi-Fi Multiple Antennas | 1718 |
| 5.33.22 | Wi-Fi Channel State Information | 1719 |
| 5.33.23 | Wi-Fi Channel State Information Configure | 1720 |
| 5.33.24 | Wi-Fi HT20/40 | 1721 |
| 5.33.25 | Wi-Fi QoS | 1721 |
| 5.33.26 | Wi-Fi AMSDU | 1722 |
| 5.33.27 | Wi-Fi Fragment | 1722 |
| 5.33.28 | WPS Enrolle | 1722 |
| 5.33.29 | Wi-Fi Buffer Usage | 1722 |
| 5.33.30 | Wi-Fi Menuconfig | 1723 |
| 5.33.31 | Troubleshooting | 1726 |
| 6 | Libraries and Frameworks | 1733 |
| 6.1 | Cloud Frameworks | 1733 |
| 6.1.1 | AWS IoT | 1733 |
| 6.1.2 | Azure IoT | 1733 |
| 6.1.3 | Google IoT Core | 1733 |
| 7 | Contributions Guide | 1735 |
| 7.1 | How to Contribute | 1735 |
| 7.2 | Before Contributing | 1735 |
| 7.3 | Pull Request Process | 1735 |
| 7.4 | Legal Part | 1735 |
| 7.5 | Related Documents | 1736 |
| 7.5.1 | Espressif IoT Development Framework Style Guide | 1736 |
| 7.5.2 | Documenting Code | 1741 |
| 7.5.3 | Documentation Add-ons and Extensions Reference | 1748 |
| 7.5.4 | Creating Examples | 1750 |

| | | |
|-----------|--|-------------|
| 7.5.5 | API Documentation Template | 1751 |
| 7.5.6 | Contributor Agreement | 1752 |
| 8 | ESP-IDF Versions | 1755 |
| 8.1 | Releases | 1755 |
| 8.2 | Which Version Should I Start With? | 1756 |
| 8.3 | Versioning Scheme | 1756 |
| 8.4 | Support Periods | 1757 |
| 8.5 | Checking the Current Version | 1757 |
| 8.6 | Git Workflow | 1758 |
| 8.7 | Updating ESP-IDF | 1758 |
| 8.7.1 | Updating to Stable Release | 1759 |
| 8.7.2 | Updating to a Pre-Release Version | 1759 |
| 8.7.3 | Updating to Master Branch | 1759 |
| 8.7.4 | Updating to a Release Branch | 1760 |
| 9 | Resources | 1761 |
| 10 | Copyrights and Licenses | 1763 |
| 10.1 | Software Copyrights | 1763 |
| 10.1.1 | Firmware Components | 1763 |
| 10.1.2 | Build Tools | 1764 |
| 10.2 | ROM Source Code Copyrights | 1764 |
| 10.3 | Xtensa libhal MIT License | 1764 |
| 10.4 | TinyBasic Plus MIT License | 1765 |
| 10.5 | TJpgDec License | 1765 |
| 11 | About | 1767 |
| 12 | Switch Between Languages/切换语言 | 1769 |
| | Index | 1771 |
| | Index | 1771 |

This is the documentation for Espressif IoT Development Framework ([esp-idf](#)). ESP-IDF is the official development framework for the [ESP32](#) chip.

| | | |
|--|--|---|
|  |  |  |
| Get Started | API Reference | H/W Reference |
|  |  |  |
| API Guides | Contribute | Resources |

Chapter 1

Get Started

This document is intended to help you set up the software development environment for the hardware based on the ESP32 chip by Espressif.

After that, a simple example will show you how to use ESP-IDF (Espressif IoT Development Framework) for menu configuration, then building, and flashing firmware onto an ESP32 board.

Note: This is documentation for stable version v4.1.3 of ESP-IDF. Other *ESP-IDF Versions* are also available.

1.1 Introduction

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

- Wi-Fi (2.4 GHz band)
- Bluetooth 4.2
- Dual high performance cores
- Ultra Low Power co-processor
- Several peripherals

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

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

1.2 What You Need

Hardware:

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

Software:

- **Toolchain** to compile code for ESP32
- **Build tools** - CMake and Ninja to build a full **Application** for ESP32
- **ESP-IDF** that essentially contains API (software libraries and source code) for ESP32 and scripts to operate the **Toolchain**
- **Text editor** to write programs (**Projects**) in C, e.g., [Eclipse](#)

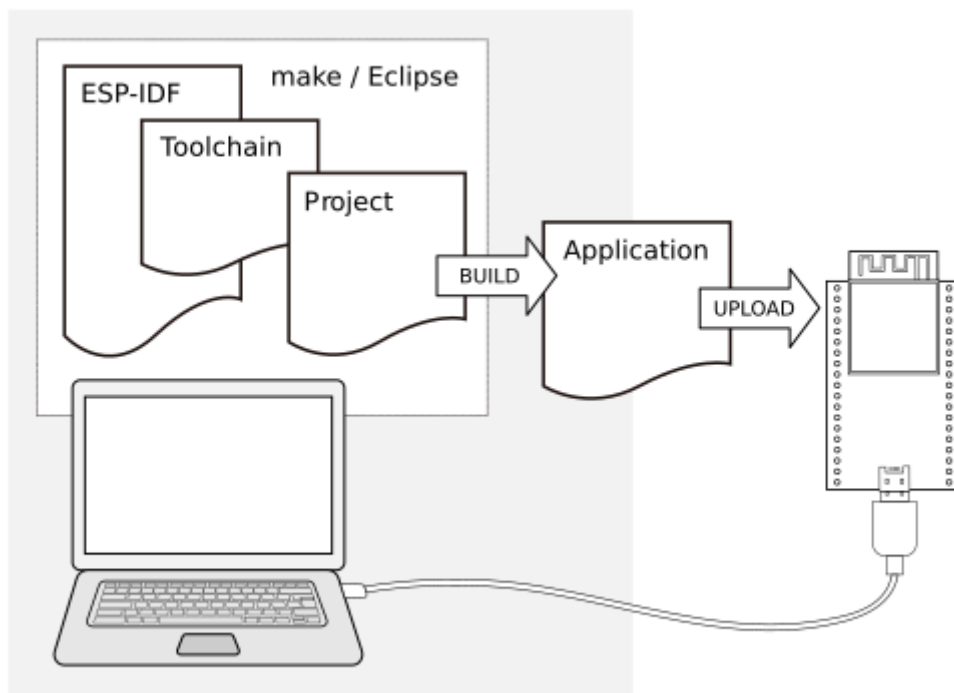


Fig. 1: Development of applications for ESP32

1.3 Development Board Overviews

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

1.3.1 ESP32-DevKitC V4 Getting Started Guide

This guide shows how to start using the ESP32-DevKitC V4 development board. For description of other versions of ESP32-DevKitC check [ESP32 Hardware Reference](#).

What You Need

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

You can skip the introduction sections and go directly to Section [Start Application Development](#).

Overview

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

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

- different ESP32 modules
 - [ESP32-WROOM-32](#)
 - [ESP32-WROOM-32D](#)
 - [ESP32-WROOM-32U](#)

- *ESP32-SOLO-1*
- *ESP32-WROVER*
- *ESP32-WROVER-B*
- *ESP32-WROVER-I*
- *ESP32-WROVER-B (IPEX)*
- male or female pin headers.

For details please refer to [Espressif Product Ordering Information](#).

Functional Description

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

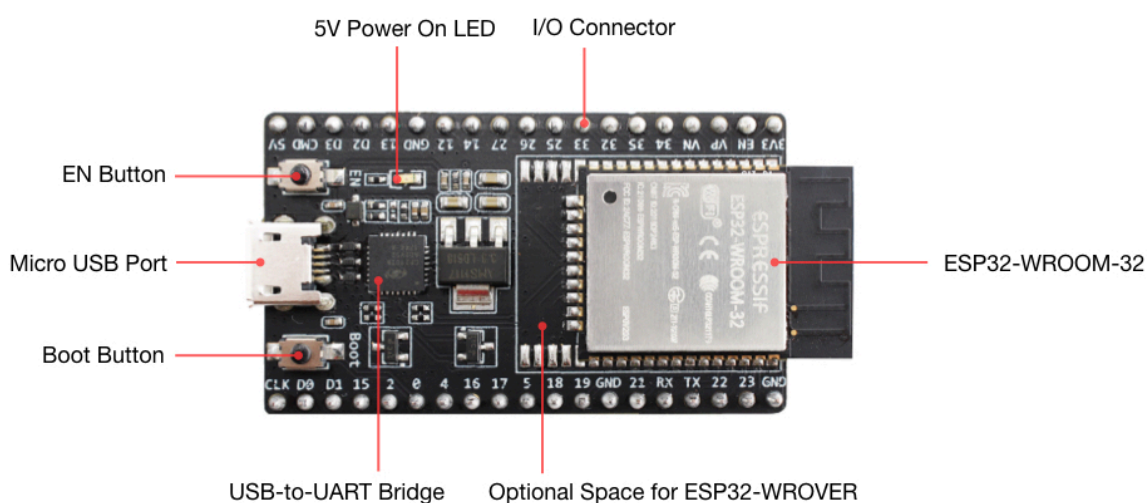


Fig. 2: ESP32-DevKitC V4 with ESP32-WROOM-32 module soldered

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

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

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

Power Supply Options

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

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

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

Note on C15

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

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

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

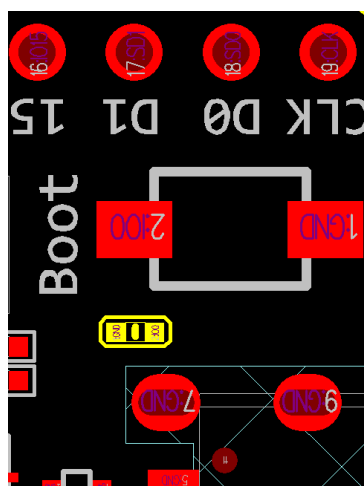


Fig. 3: Location of C15 (yellow) on ESP32-DevKitC V4 board

Start Application Development

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

After that, proceed to [ESP32 Hardware Reference](#), where Section *Installation Step by Step* will quickly help you set up the development environment and then flash an example project onto your board.

Board Dimensions

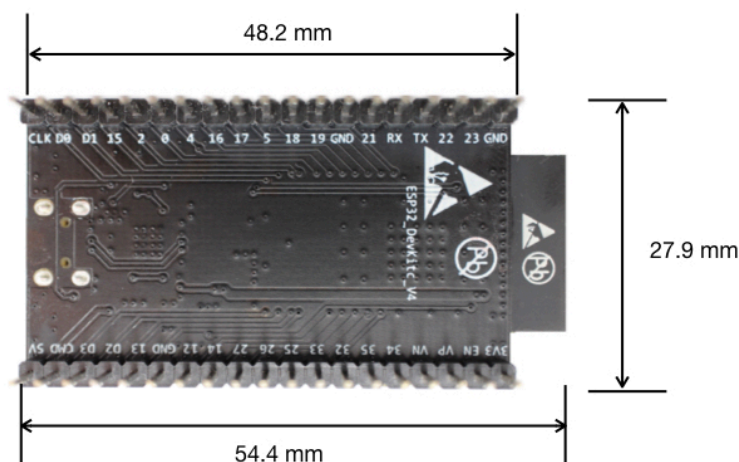


Fig. 4: ESP32 DevKitC board dimensions - back

Related Documents

- [ESP32-DevKitC V4 schematics \(PDF\)](#)
- [ESP32 Datasheet \(PDF\)](#)
- [ESP32-WROOM-32 Datasheet \(PDF\)](#)
- [ESP32-WROOM-32D & ESP32-WROOM-32U Datasheet \(PDF\)](#)
- [ESP32-WROVER Datasheet \(PDF\)](#)
- [ESP32-WROVER-B Datasheet \(PDF\)](#)
- [Espressif Product Ordering Information \(PDF\)](#)

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

What You Need

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

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

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

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

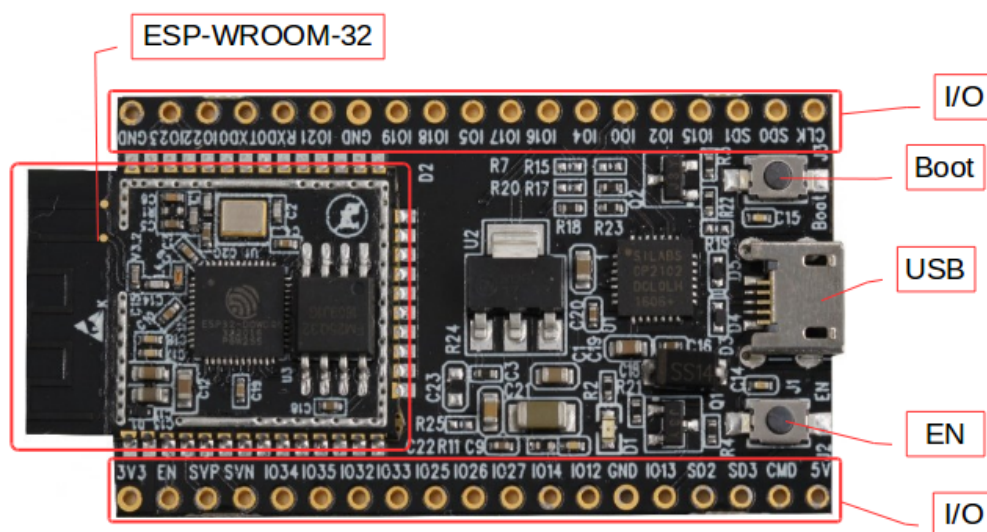


Fig. 5: ESP32-DevKitC V2 board layout

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

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

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

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

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

After that, proceed to [ESP32 Hardware Reference](#), where Section *Installation Step by Step* will quickly help you set up the development environment and then flash an example project onto your board.

Related Documents

- [ESP32-DevKitC schematics \(PDF\)](#)
- [ESP32 Datasheet \(PDF\)](#)
- [ESP32-WROOM-32 Datasheet \(PDF\)](#)

1.3.2 ESP-WROVER-KIT V4.1 Getting Started Guide

This guide shows how to get started with the ESP-WROVER-KIT V4.1 development board and also provides information about its functionality and configuration options. For the description of other ESP-WROVER-KIT versions, please check [ESP32 Hardware Reference](#).

What You Need

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

You can skip the introduction sections and go directly to Section [Start Application Development](#).

Overview

ESP-WROVER-KIT is an ESP32-based development board produced by [Espressif](#).

ESP-WROVER-KIT features the following integrated components:

- ESP32-WROVER-B module
- LCD screen
- MicroSD card slot

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

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

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

Functionality Overview

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

Functional Description

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

The table below provides description in the following manner:

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

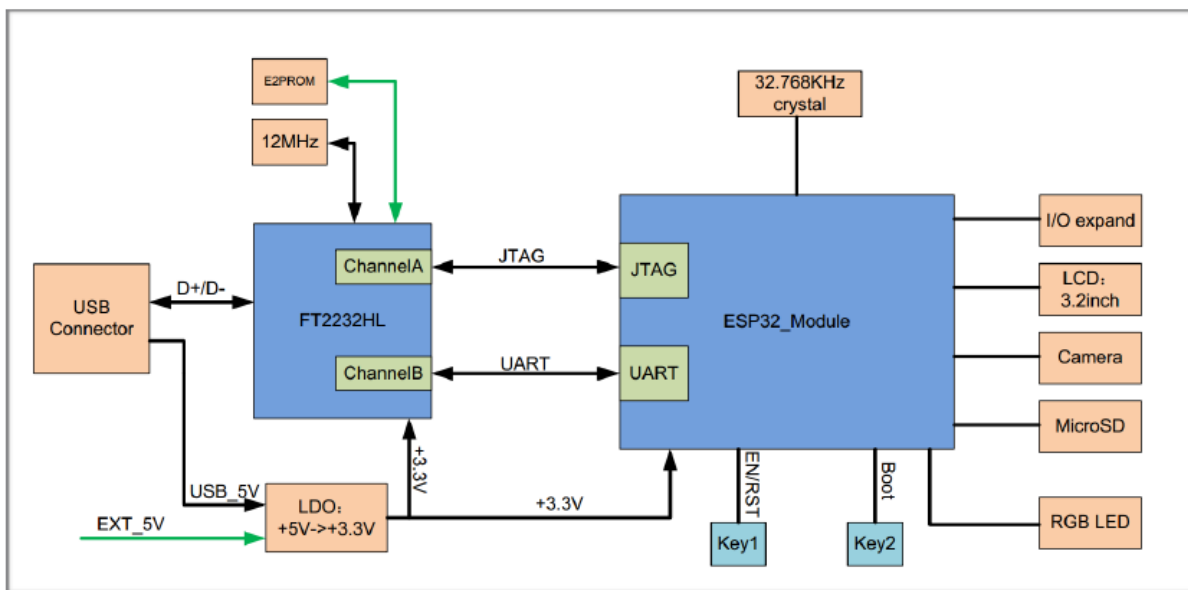


Fig. 6: ESP-WROVER-KIT block diagram

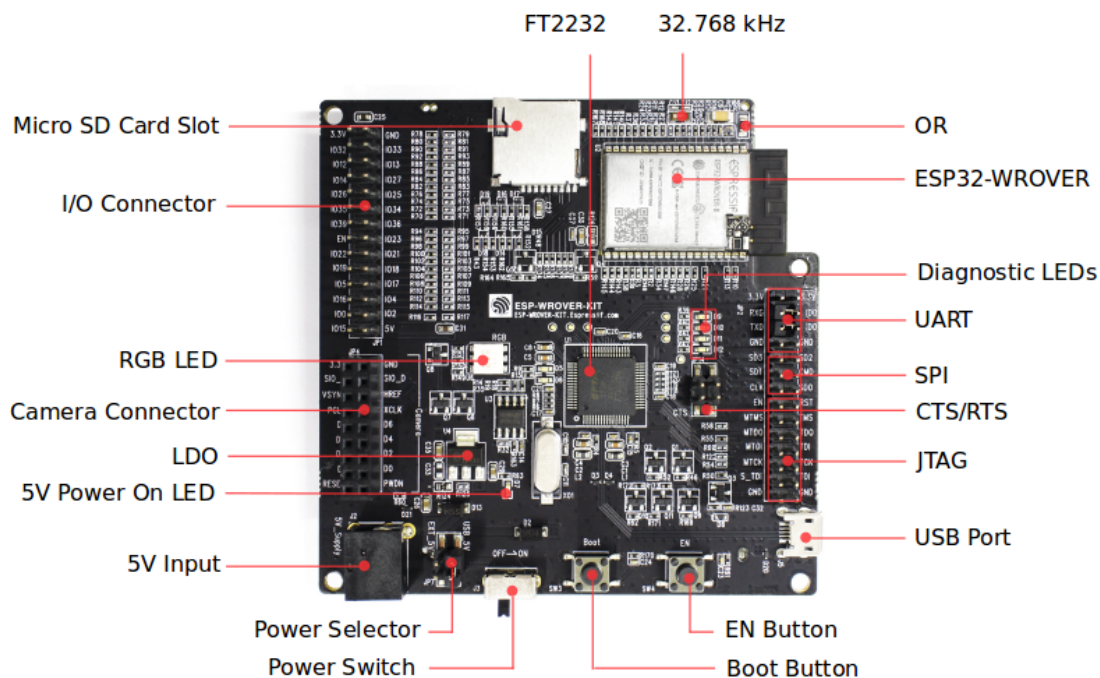


Fig. 7: ESP-WROVER-KIT board layout - front

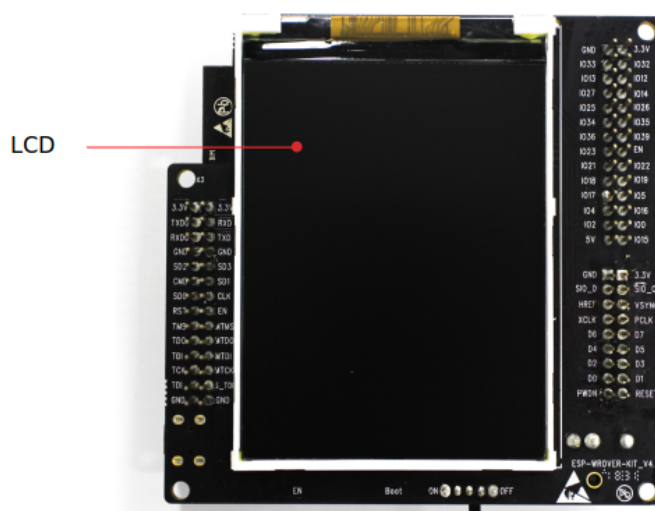
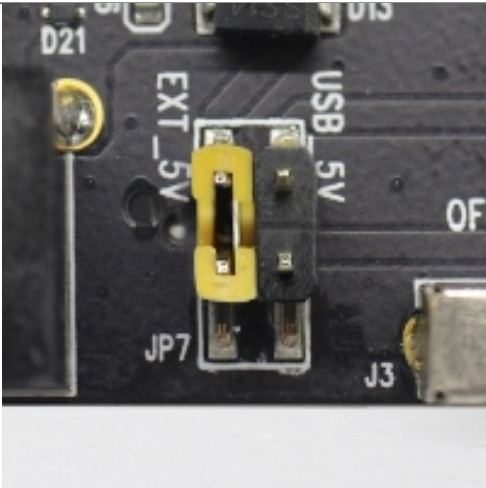
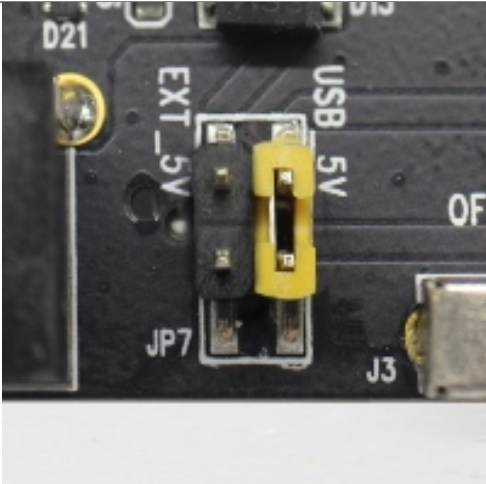
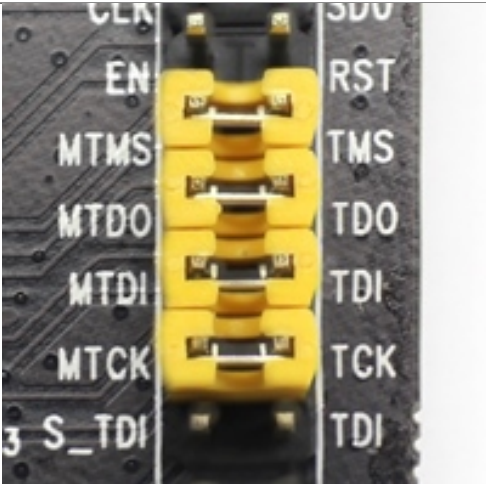
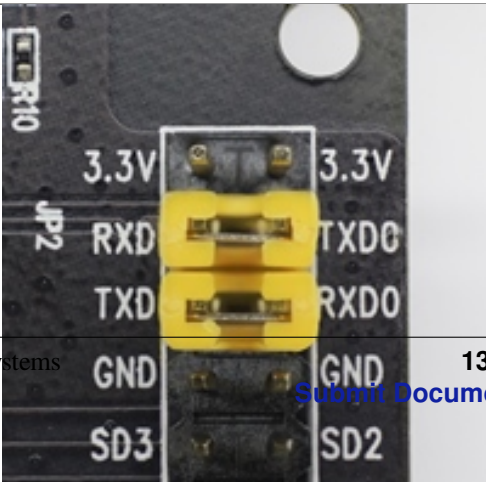


Fig. 8: ESP-WROVER-KIT board layout - back

| Key Component | Description |
|-----------------|--|
| FT2232 | The FT2232 chip serves as a multi-protocol USB-to-serial bridge which can be programmed and controlled via USB to provide communication with ESP32. FT2232 also features USB-to-JTAG interface which is available on channel A of the chip, while USB-to-serial is on channel B. The FT2232 chip enhances user-friendliness in terms of application development and debugging. See ESP-WROVER-KIT V4.1 schematic . |
| 32.768 kHz | External precision 32.768 kHz crystal oscillator serves as a clock with low-power consumption while the chip is in Deep-sleep mode. |
| 0R | Zero-ohm resistor intended as a placeholder for a current shunt, can be desoldered or replaced with a current shunt to facilitate the measurement of ESP32's current consumption in different modes. |
| ESP32-WROVER-B | This ESP32 module features 64-Mbit PSRAM for flexible extended storage and data processing capabilities. |
| Diagnostic LEDs | Four red LEDs connected to the GPIO pins of FT2232. Intended for future use. |
| UART | Serial port. The serial TX/RX signals of FT2232 and ESP32 are broken out to the inward and outward sides of JP2 respectively. By default, these pairs of pins are connected with jumpers. To use ESP32's serial interface, remove the jumpers and connect another external serial device to the respective pins. |
| SPI | By default, ESP32 uses its SPI interface to access flash and PSRAM memory inside the module. Use these pins to connect ESP32 to another SPI device. In this case, an extra chip select (CS) signal is needed. Please note that the voltage of this interface is 3.3V. |
| CTS/RTS | Serial port flow control signals: the pins are not connected to the circuitry by default. To enable them, short the respective pins of JP14 with jumpers. |
| JTAG | JTAG interface. JTAG signals of FT2232 and ESP32 are broken out to the inward and outward sides of JP2 respectively. By default, these pairs of pins are disconnected. To enable JTAG, short the respective pins with jumpers as shown in Section Setup Options . |
| USB Port | USB interface. Power supply for the board as well as the communication interface between a computer and the board. |
| EN Button | Reset button. |
| Boot Button | Download button. Holding down Boot and then pressing EN initiates Firmware Download mode for downloading firmware through the serial port. |

Setup Options

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

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

Allocation of ESP32 Pins

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

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

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

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

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

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

Legend:

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

32.768 kHz Oscillator

| . | ESP32 Pin |
|---|-----------|
| 1 | GPIO32 |
| 2 | GPIO33 |

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

SPI Flash / JP2

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

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

JTAG / JP2

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

Camera / JP4

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

- Signals D0 .. D7 denote camera data bus

RGB LED

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

MicroSD Card

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

LCD / U5

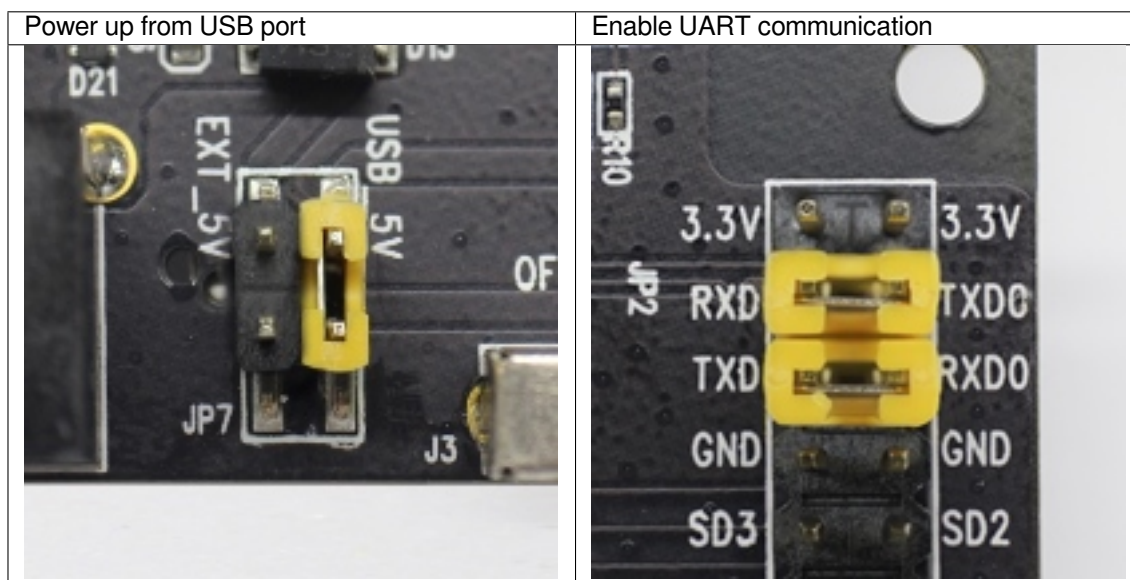
| . | ESP32 Pin | LCD Signal |
|---|-----------|------------|
| 1 | GPIO18 | RESET |
| 2 | GPIO19 | SCL |
| 3 | GPIO21 | D/C |
| 4 | GPIO22 | CS |
| 5 | GPIO23 | SDA |
| 6 | GPIO25 | SDO |
| 7 | GPIO5 | Backlight |

Start Application Development

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

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

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



Do not install any other jumpers.

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

Now to Development Please proceed to [ESP32 Hardware Reference](#), where Section *Installation Step by Step* will quickly help you set up the development environment and then flash an example project onto your board.

Related Documents

- [ESP-WROVER-KIT V4.1 schematic \(PDF\)](#)
- [ESP32 Datasheet \(PDF\)](#)
- [ESP32-WROVER-B Datasheet \(PDF\)](#)
- [JTAG Debugging](#)
- [ESP32 Hardware Reference](#)

ESP-WROVER-KIT V3 Getting Started Guide This guide shows how to get started with the ESP-WROVER-KIT V3 development board and also provides information about its functionality and configuration options. For the description of other ESP-WROVER-KIT versions, please check [ESP32 Hardware Reference](#).

What You Need

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

You can skip the introduction sections and go directly to Section [Start Application Development](#).

Overview ESP-WROVER-KIT is an ESP32-based development board produced by [Espressif](#). This board features an integrated LCD screen and MicroSD card slot.

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

- [ESP32-WROOM-32](#)
- [ESP32-WROVER](#)

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

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

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

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

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

The table below provides description in the following manner:

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

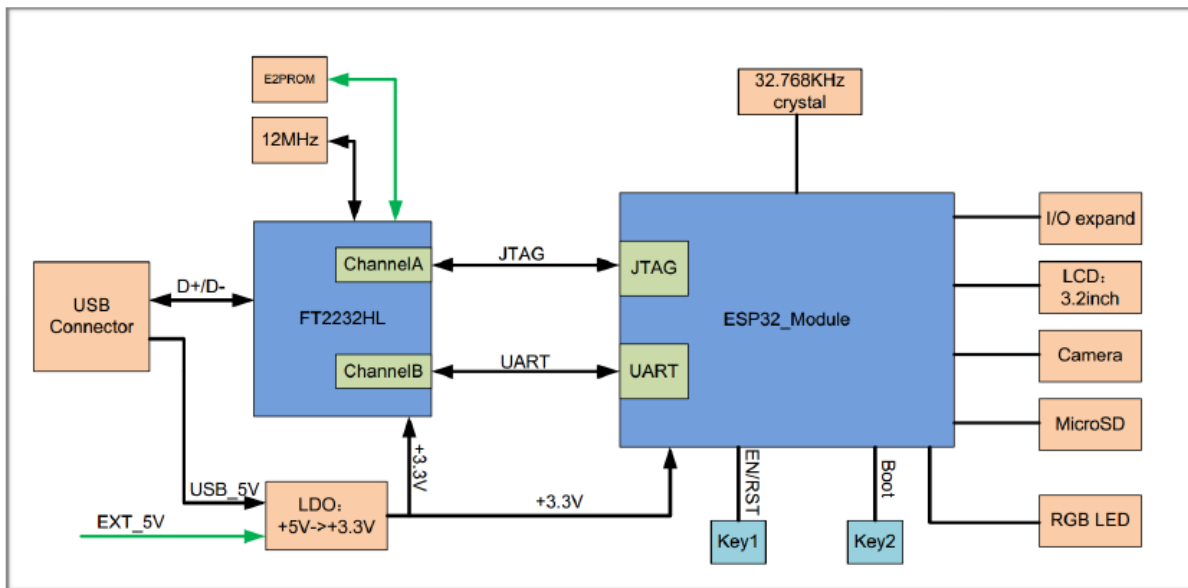


Fig. 9: ESP-WROVER-KIT block diagram

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

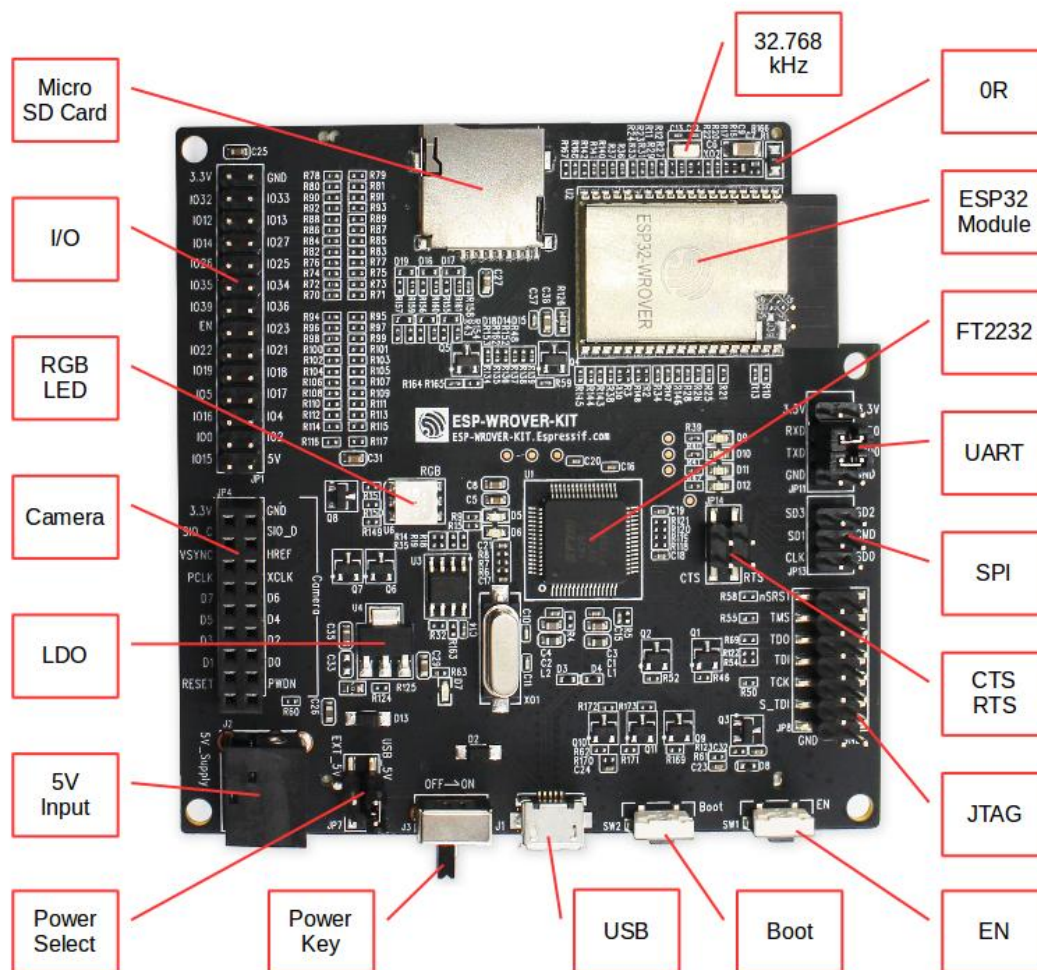


Fig. 10: ESP-WROVER-KIT board layout - front

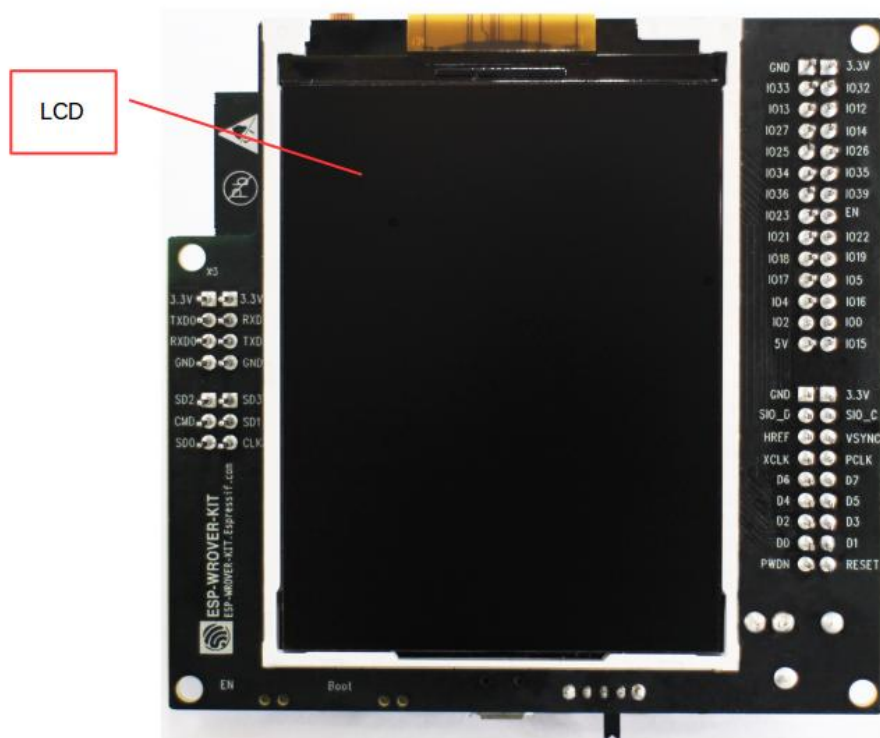
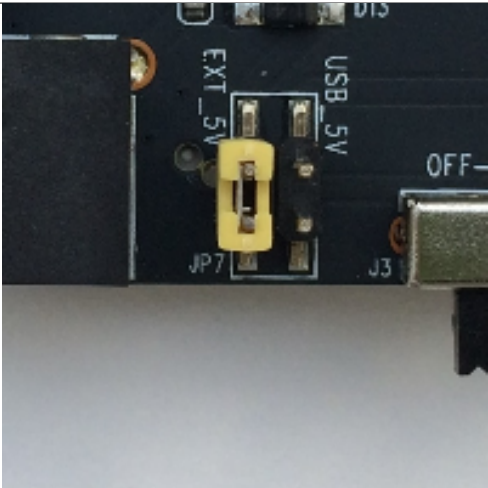
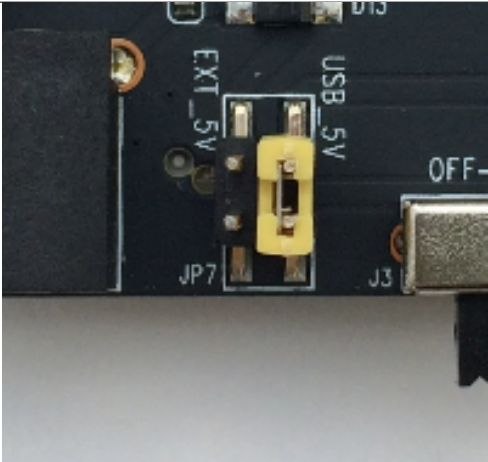
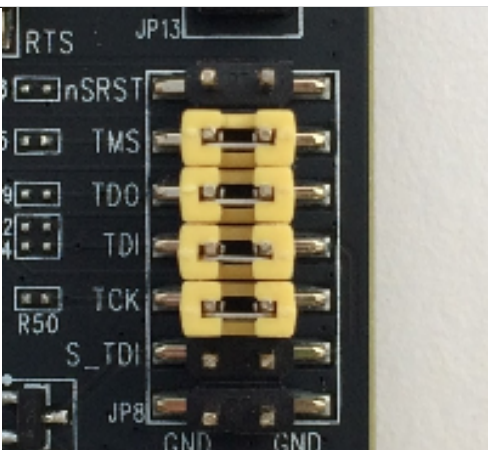
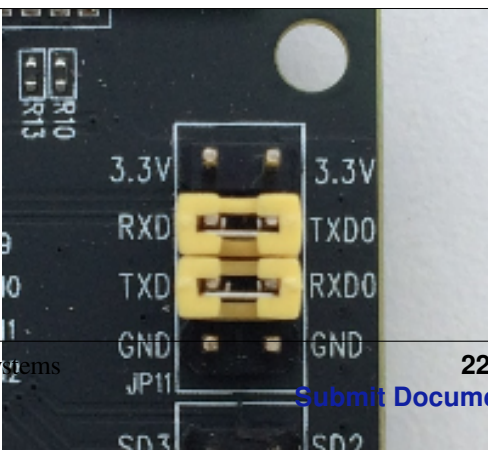


Fig. 11: ESP-WROVER-KIT board layout - back

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

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

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

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

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

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

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

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

Legend:

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

32.768 kHz Oscillator

| . | ESP32 Pin |
|---|-----------|
| 1 | GPIO32 |
| 2 | GPIO33 |

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

SPI Flash / JP13

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

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

JTAG / JP8

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

Camera / JP4

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

- Signals D0 .. D7 denote camera data bus

RGB LED

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

MicroSD Card

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

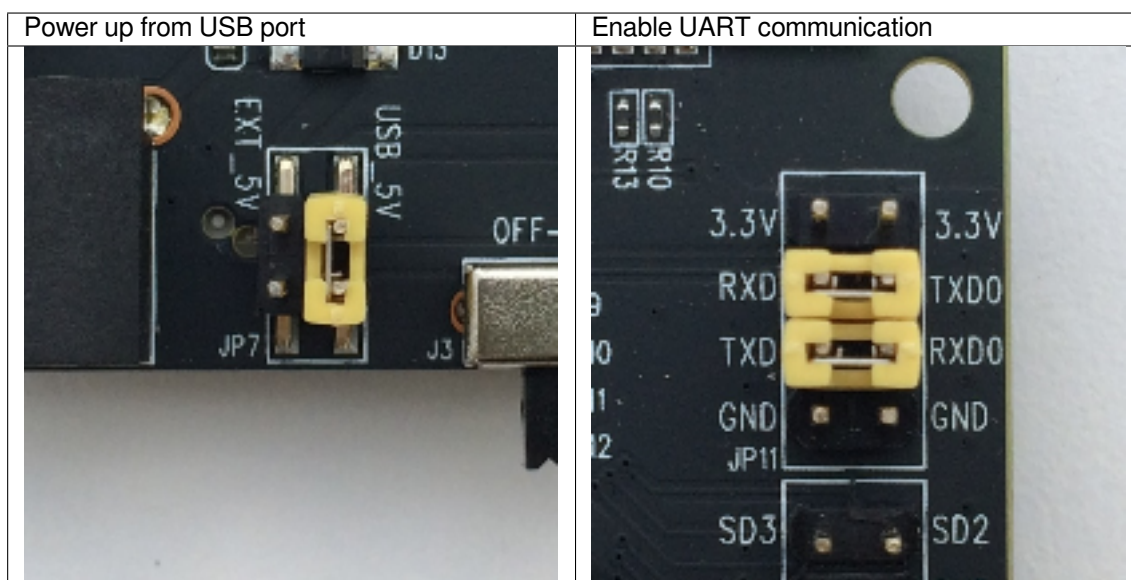
LCD / U5

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

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

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

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



Do not install any other jumpers.

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

Now to Development Please proceed to [ESP32 Hardware Reference](#), where Section *Installation Step by Step* will quickly help you set up the development environment and then flash an example project onto your board.

Related Documents

- [ESP-WROVER-KIT V3 schematic \(PDF\)](#)
- [ESP32 Datasheet \(PDF\)](#)
- [ESP32-WROVER Datasheet \(PDF\)](#)
- [ESP32-WROOM-32 Datasheet \(PDF\)](#)
- [JTAG Debugging](#)
- [ESP32 Hardware Reference](#)

ESP-WROVER-KIT V2 Getting Started Guide This guide shows how to get started with the ESP-WROVER-KIT V2 development board and also provides information about its functionality and configuration options. For the description of other ESP-WROVER-KIT versions, please check [ESP32 Hardware Reference](#).

What You Need

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

You can skip the introduction sections and go directly to Section [Start Application Development](#).

Overview ESP-WROVER-KIT is an ESP32-based development board produced by [Espressif](#). This board features an integrated LCD screen and MicroSD card slot.

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

- [ESP32-WROOM-32](#)
- [ESP32-WROVER](#)

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

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

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

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

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

The table below provides description in the following manner:

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

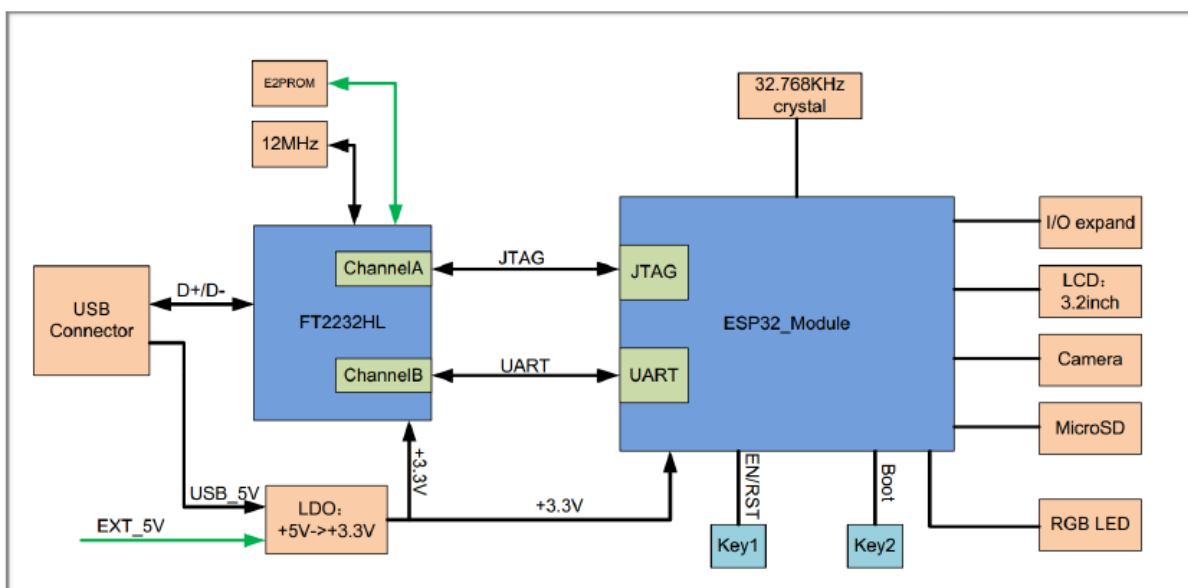


Fig. 12: ESP-WROVER-KIT block diagram

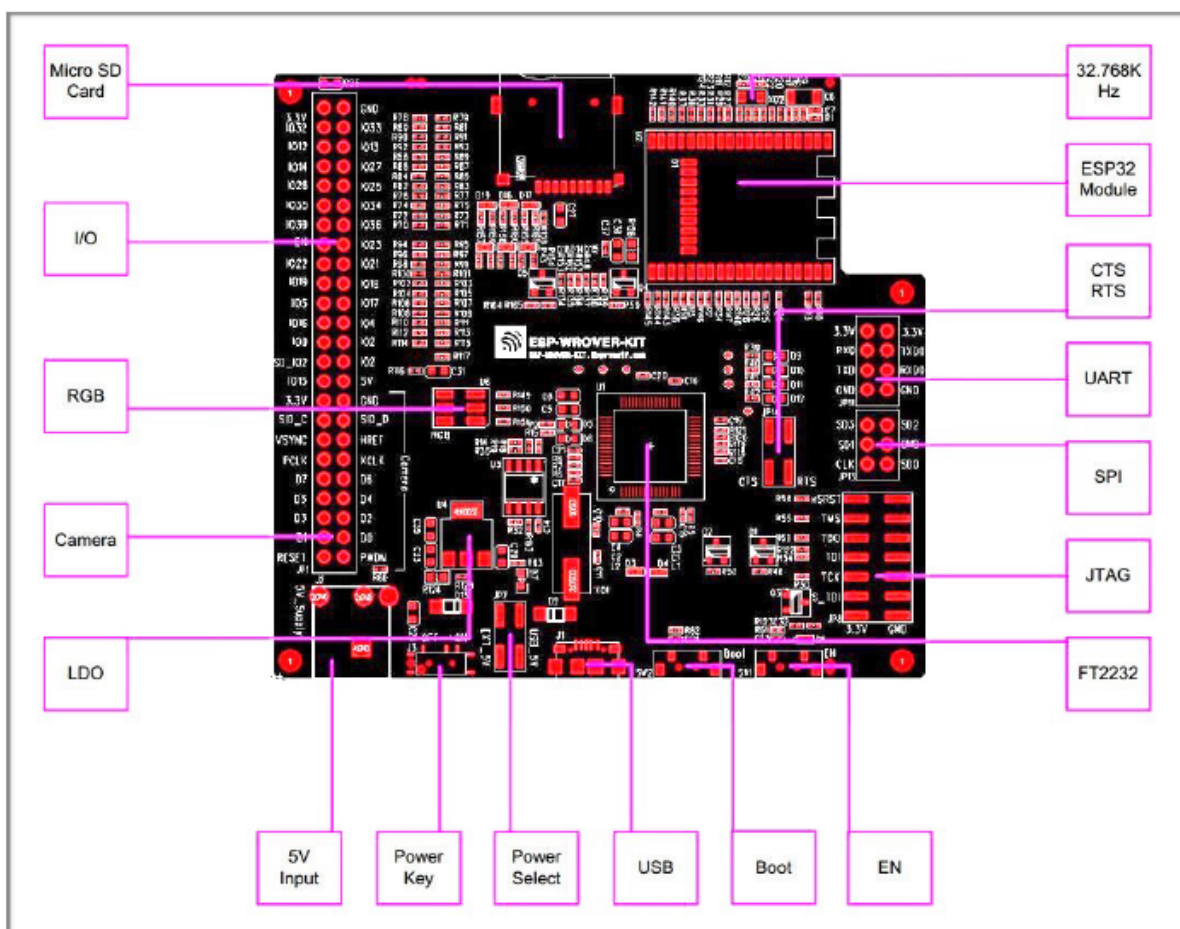


Fig. 13: ESP-WROVER-KIT board layout - front

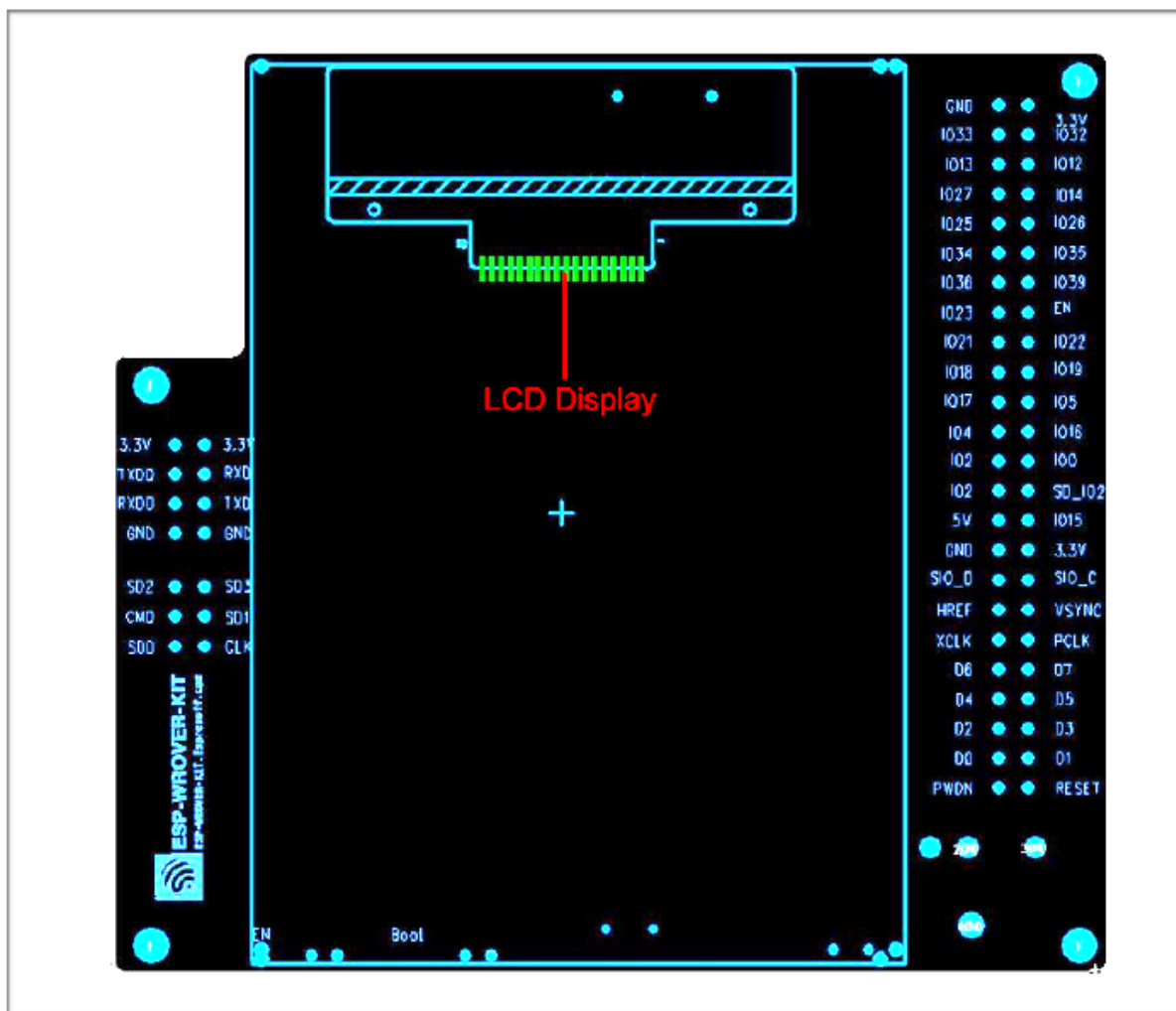
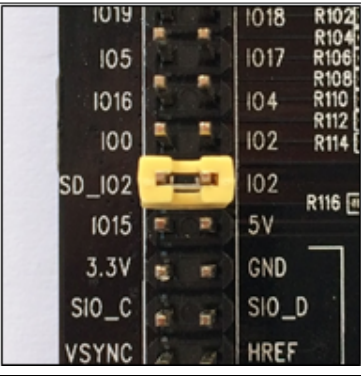
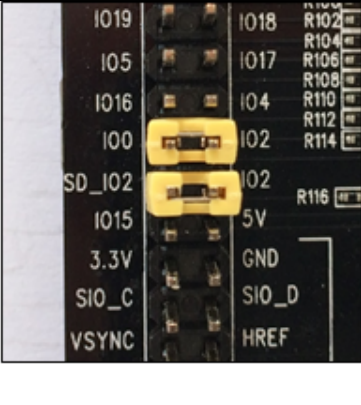
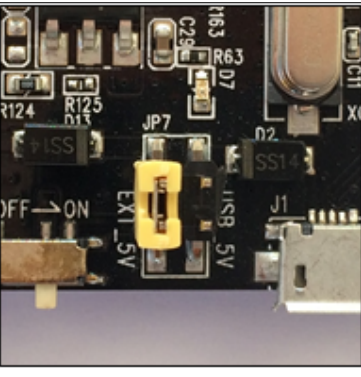
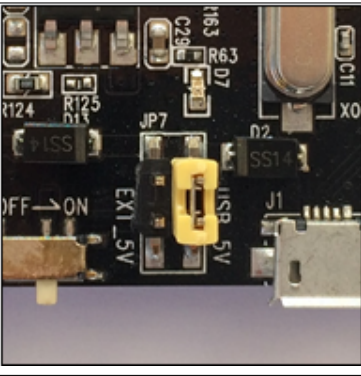
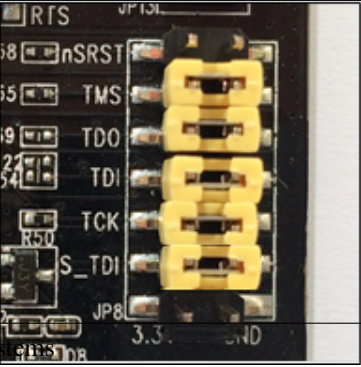


Fig. 14: ESP-WROVER-KIT board layout - back

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

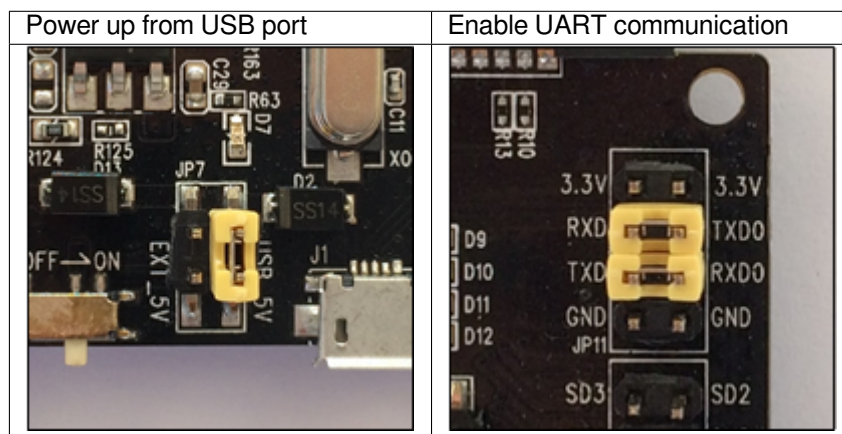
Setup Options There are five jumper blocks available to set up the board functionality. The most frequently required options are listed in the table below.

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

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

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

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



Do not install any other jumpers.

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

Now to Development Please proceed to *ESP32 Hardware Reference*, where Section *Installation Step by Step* will quickly help you set up the development environment and then flash an example project onto your board.

Related Documents

- [ESP-WROVER-KIT V2 schematic \(PDF\)](#)
- [ESP32 Datasheet \(PDF\)](#)
- [ESP32-WROVER Datasheet \(PDF\)](#)
- [ESP32-WROOM-32 Datasheet \(PDF\)](#)
- [JTAG Debugging](#)
- [ESP32 Hardware Reference](#)

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

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

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

What You Need

- [ESP32-PICO-KIT mini development board](#)
- USB 2.0 A to Micro B cable
- Computer running Windows, Linux, or macOS

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

Overview

ESP32-PICO-KIT is an ESP32-based mini development board produced by [Espressif](#).

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

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

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

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

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

Note:

1. The 2 x 3 pads not populated with pin headers are connected to the flash memory embedded in the ESP32-PICO-D4 SiP module. For more details see module's datasheet in [Related Documents](#).
2. ESP32-PICO-KIT comes with male headers by default.

Functionality Overview

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

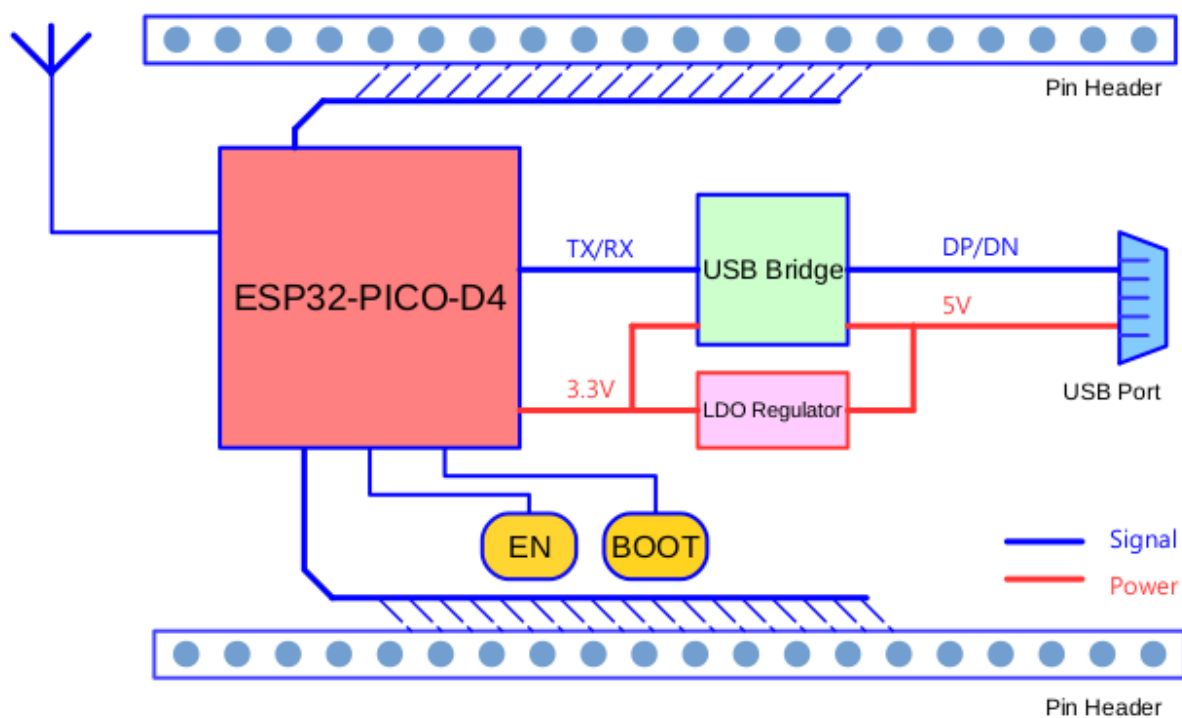


Fig. 15: ESP32-PICO-KIT block diagram

Functional Description

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

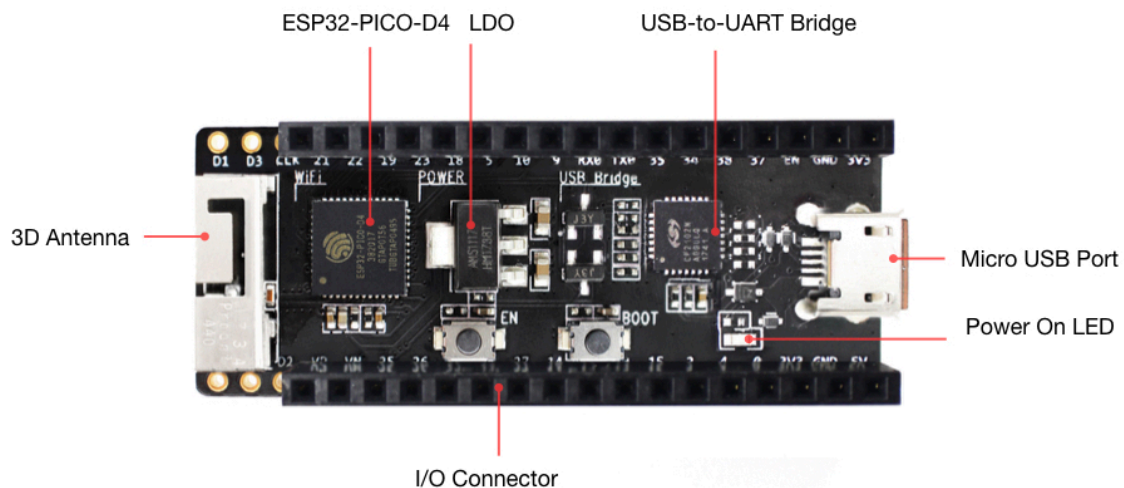


Fig. 16: ESP32-PICO-KIT board layout

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

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

Power Supply Options

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

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

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

Pin Descriptions

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

Header J2

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

Header J3

| No. | Name | Type | Function |
|-------------------|------------------|--------|--|
| 1 | FLASH_CS (FCS) | I/O | GPIO16, HS1_DATA4 (<i>See 1</i>), U2RXD, EMAC_CLK_OUT |
| 2 | FLASH_SD0 (FSD0) | I/O | GPIO17, HS1_DATA5 (<i>See 1</i>), U2TXD, EMAC_CLK_OUT_180 |
| 3 | FLASH_SD2 (FSD2) | I/O | GPIO11, SD_CMD, SPICS0, HS1_CMD (<i>See 1</i>), U1RTS |
| 4 | SENSOR_VP (FSVP) | I | GPIO36, ADC1_CH0, RTC_GPIO0 |
| 5 | SENSOR_VN (FSVN) | I | GPIO39, ADC1_CH3, RTC_GPIO3 |
| 6 | IO25 | I/O | GPIO25, DAC_1, ADC2_CH8, RTC_GPIO6, EMAC_RXD0 |
| 7 | IO26 | I/O | GPIO26, DAC_2, ADC2_CH9, RTC_GPIO7, EMAC_RXD1 |
| 8 | IO32 | I/O | 32K_XP (<i>See 2a</i>), ADC1_CH4, TOUCH9, RTC_GPIO9 |
| 9 | IO33 | I/O | 32K_XN (<i>See 2b</i>), ADC1_CH5, TOUCH8, RTC_GPIO8 |
| 10 | IO27 | I/O | GPIO27, ADC2_CH7, TOUCH7, RTC_GPIO17 EMAC_RX_DV |
| Espressif Systems | IO14 | 37 I/O | Release v4.1.3 ADC2_CH6, TOUCH6, RTC_GPIO16, MTMS, HSPICLK, |

The following notes give more information about the items in the tables above.

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

Start Application Development

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

After that, proceed to [ESP32 Hardware Reference](#), where Section *Installation Step by Step* will quickly help you set up the development environment and then flash an example project onto your board.

Board Dimensions

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

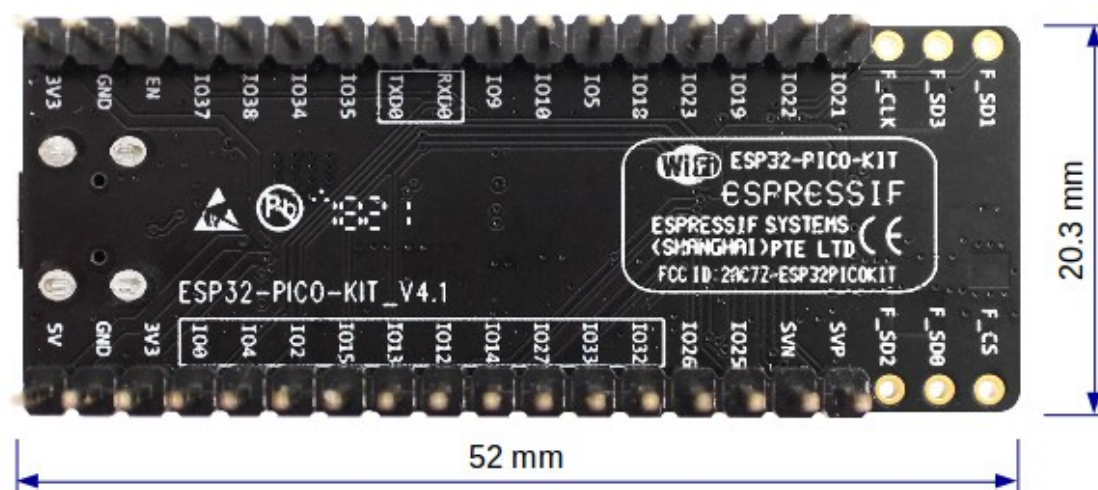


Fig. 17: ESP32-PICO-KIT dimensions - back

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

Related Documents

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

ESP32-PICO-KIT V3 Getting Started Guide This guide shows how to get started with the ESP32-PICO-KIT V3 mini development board. For the description of other ESP32-PICO-KIT versions, please check [ESP32 Hardware Reference](#).

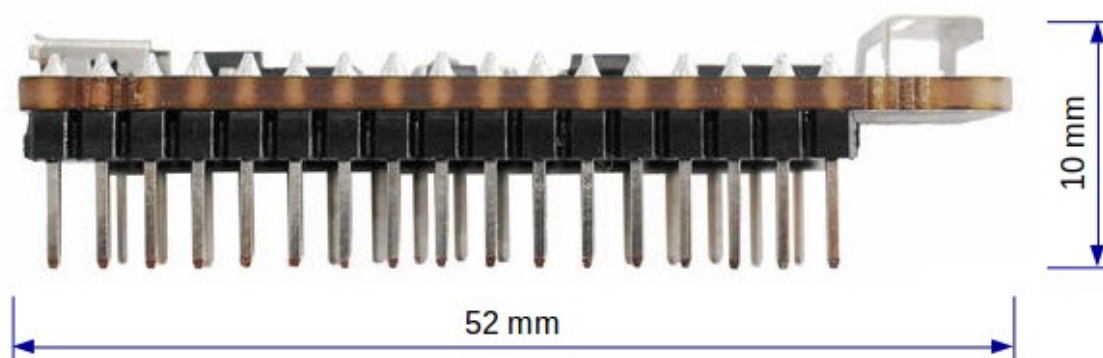


Fig. 18: ESP32-PICO-KIT dimensions - side

What You Need

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

You can skip the introduction sections and go directly to Section [Start Application Development](#).

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

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

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

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

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

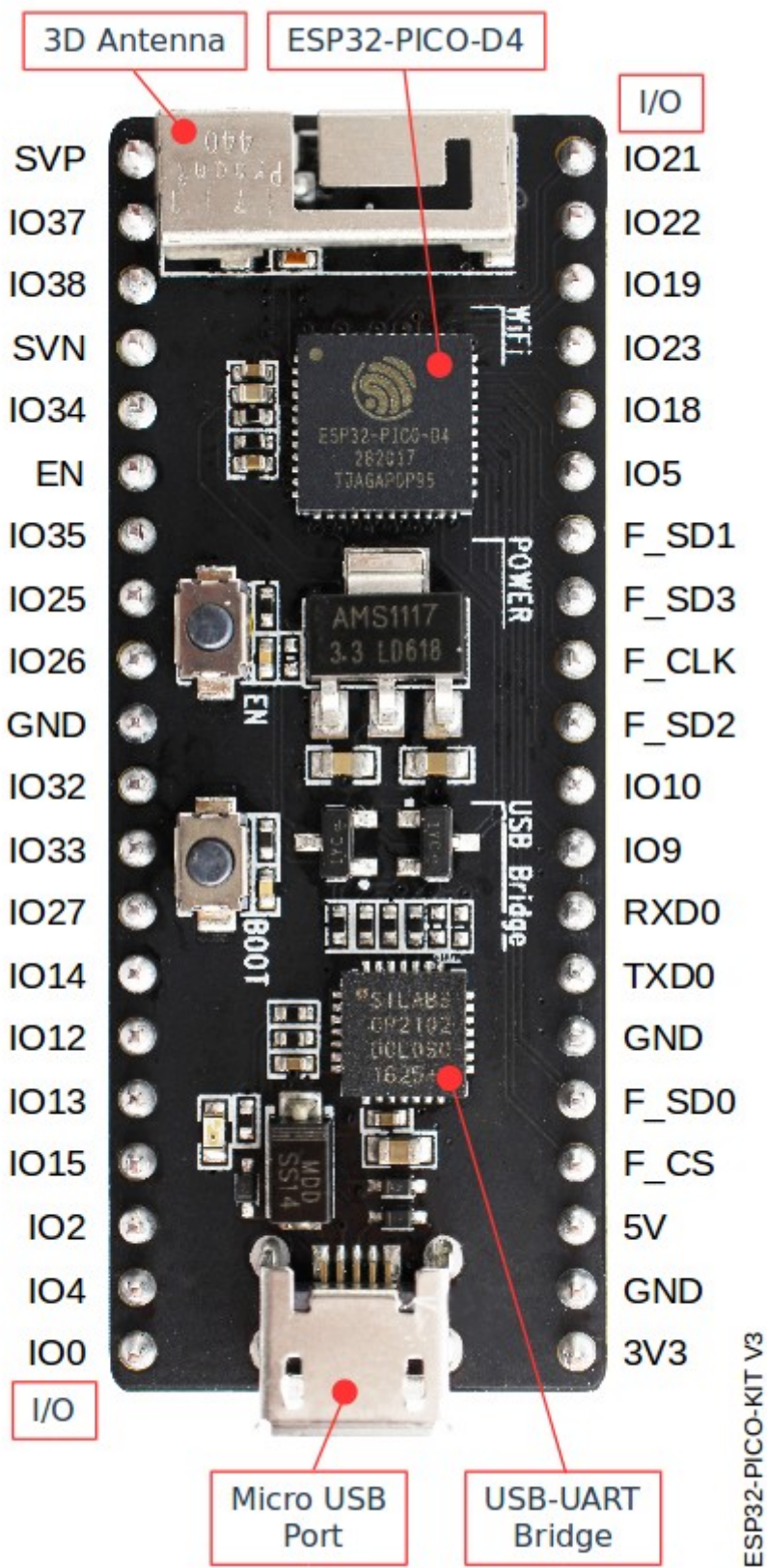


Fig. 19: ESP32-PICO-KIT V3 board layout

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

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

After that, proceed to [ESP32 Hardware Reference](#), where Section *Installation Step by Step* will quickly help you set up the development environment and then flash an example project onto your board.

Related Documents

- [ESP32-PICO-KIT V3 schematic \(PDF\)](#)
- [ESP32-PICO-D4 Datasheet \(PDF\)](#)
- [ESP32 Hardware Reference](#)

1.3.4 ESP32-Ethernet-Kit V1.1 Getting Started Guide

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

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

What You Need

- [ESP32-Ethernet-Kit V1.1 board](#)
- USB 2.0 A to Micro B Cable
- Computer running Windows, Linux, or macOS

You can skip the introduction sections and go directly to Section [Start Application Development](#).

Overview

ESP32-Ethernet-Kit is an ESP32-based development board produced by [Espressif](#).

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

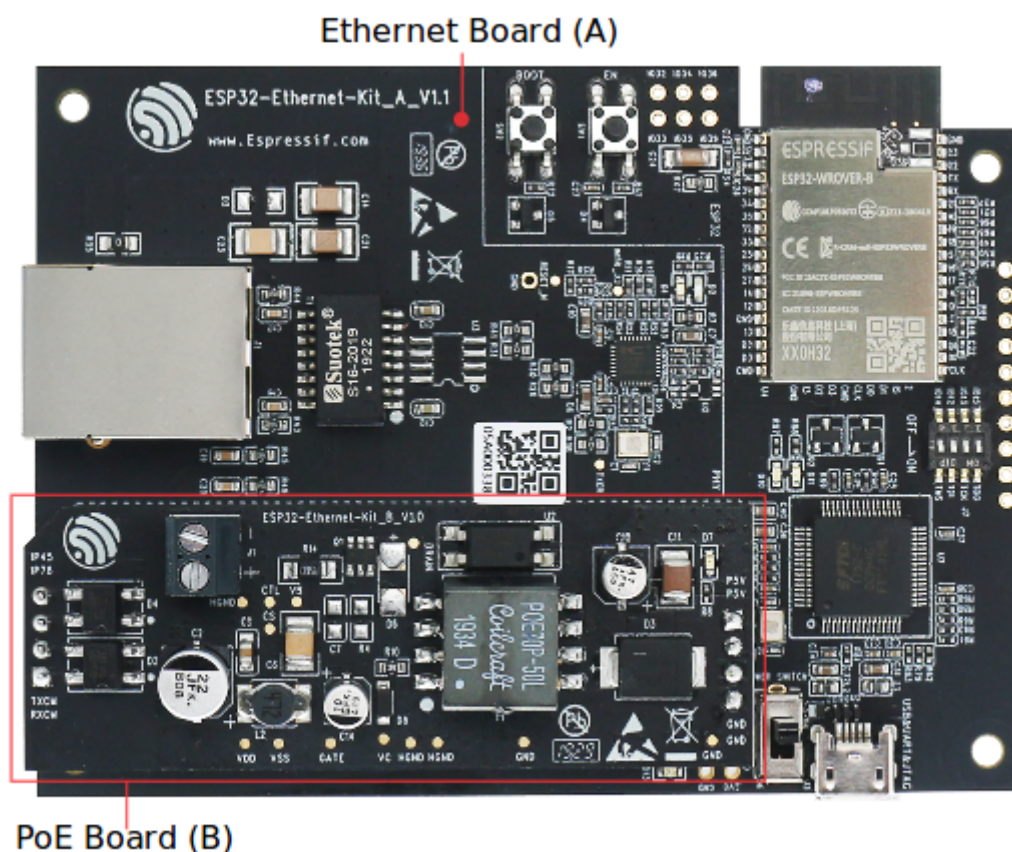


Fig. 20: ESP32-Ethernet-Kit V1.1

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

Functionality Overview

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

Functional Description

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

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

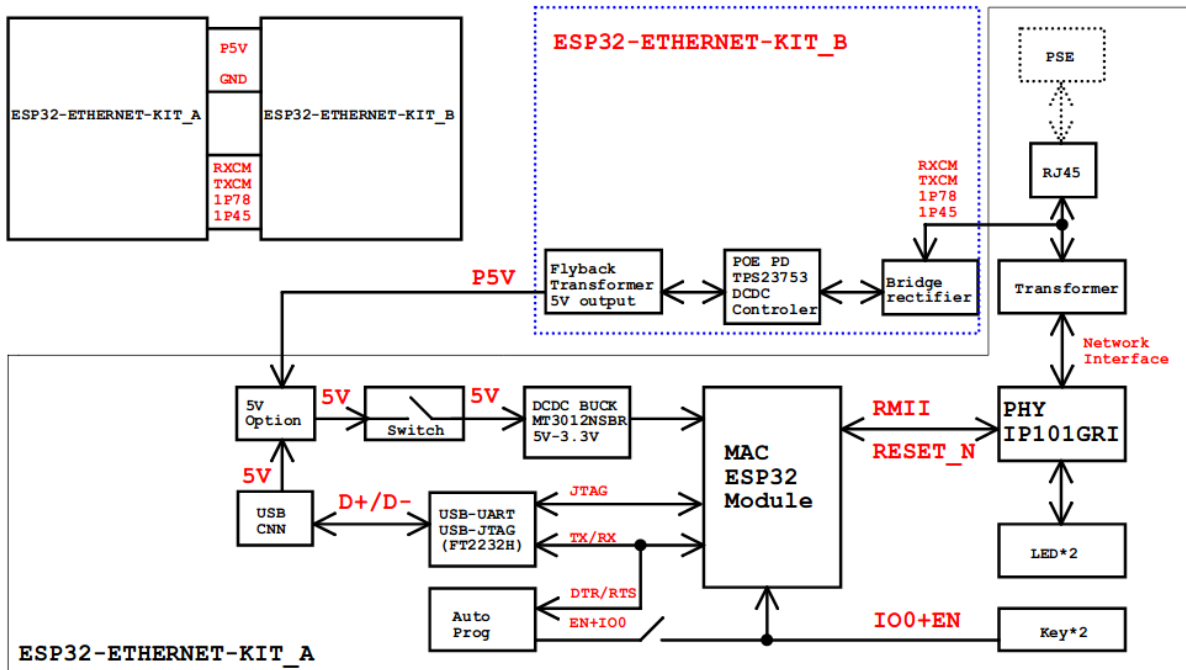


Fig. 21: ESP32-Ethernet-Kit block diagram (click to enlarge)

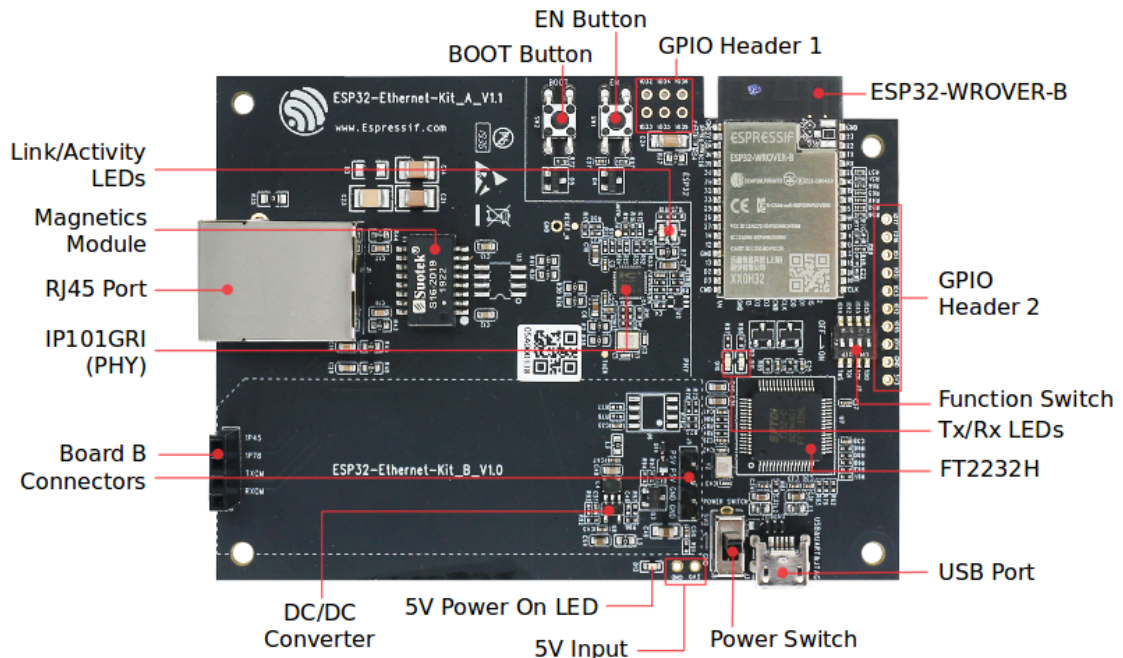


Fig. 22: ESP32-Ethernet-Kit - Ethernet board (A) layout (click to enlarge)

| Key Component | Description |
|--------------------|--|
| ESP32-WROVER-B | This ESP32 module features 64-Mbit PSRAM for flexible extended storage and data processing capabilities. |
| GPIO Header 2 | Five unpopulated through-hole solder pads to provide access to selected GPIOs of ESP32. For details, see GPIO Header 2 . |
| Function Switch | A 4-bit DIP switch used to configure the functionality of selected GPIOs of ESP32. Please note that placement of GPIO pin number marking on the board's silkscreen besides the DIP switch is incorrect. For details and correct pin allocation see Function Switch . |
| Tx/Rx LEDs | Two LEDs to show the status of UART transmission. |
| FT232RL | The FT232RL chip serves as a multi-protocol USB-to-serial bridge which can be programmed and controlled via USB to provide communication with ESP32. FT232RL also features USB-to-JTAG interface which is available on channel A of the chip, while USB-to-serial is on channel B. The FT232RL chip enhances user-friendliness in terms of application development and debugging. See ESP32-Ethernet-Kit V1.1 Ethernet board (A) schematic . |
| USB Port | USB interface. Power supply for the board as well as the communication interface between a computer and the board. |
| Power Switch | Power On/Off Switch. Toggling the switch to 5V0 position powers the board on, toggling to GND position powers the board off. |
| 5V Input | The 5V power supply interface can be more convenient when the board is operating autonomously (not connected to a computer). |
| 5V Power On LED | This red LED turns on when power is supplied to the board, either from USB or 5V Input. |
| DC/DC Converter | Provided DC 5 V to 3.3 V conversion, output current up to 2A. |
| Board B Connectors | A pair male and female header pins for mounting the PoE board (B) . |
| IP101GR (PHY) | The physical layer (PHY) connection to the Ethernet cable is implemented using the IP101GR chip. The connection between PHY and ESP32 is done through the reduced media-independent interface (RMII), a variant of the media-independent interface (MII) standard. The PHY supports the IEEE 802.3 / 802.3u standard of 10/100Mbps. |
| RJ45 Port | Ethernet network data transmission port. |
| Magnetics Module | The Magnetics are part of the Ethernet specification to protect against faults and transients, including rejection of common mode signals between the transceiver IC and the cable. The magnetics also provide galvanic isolation between the transceiver and the Ethernet device. |
| Link/Activity LEDs | Two LEDs (green and red) that respectively indicate the "Link" and "Activity" statuses of the PHY. |
| BOOT Button | Download button. Holding down BOOT and then pressing EN initiates Firmware Download mode for downloading firmware through the serial port. |
| EN Button | Reset button. |
| GPIO Header 1 | This header provides six unpopulated through-hole solder pads connected to spare GPIOs of ESP32. For details, see GPIO Header 1 . |

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

The PoE board (B) has the following features:

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

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



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

Table 1: Table PoE board (B)

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

Setup Options

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

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

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

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

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

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

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

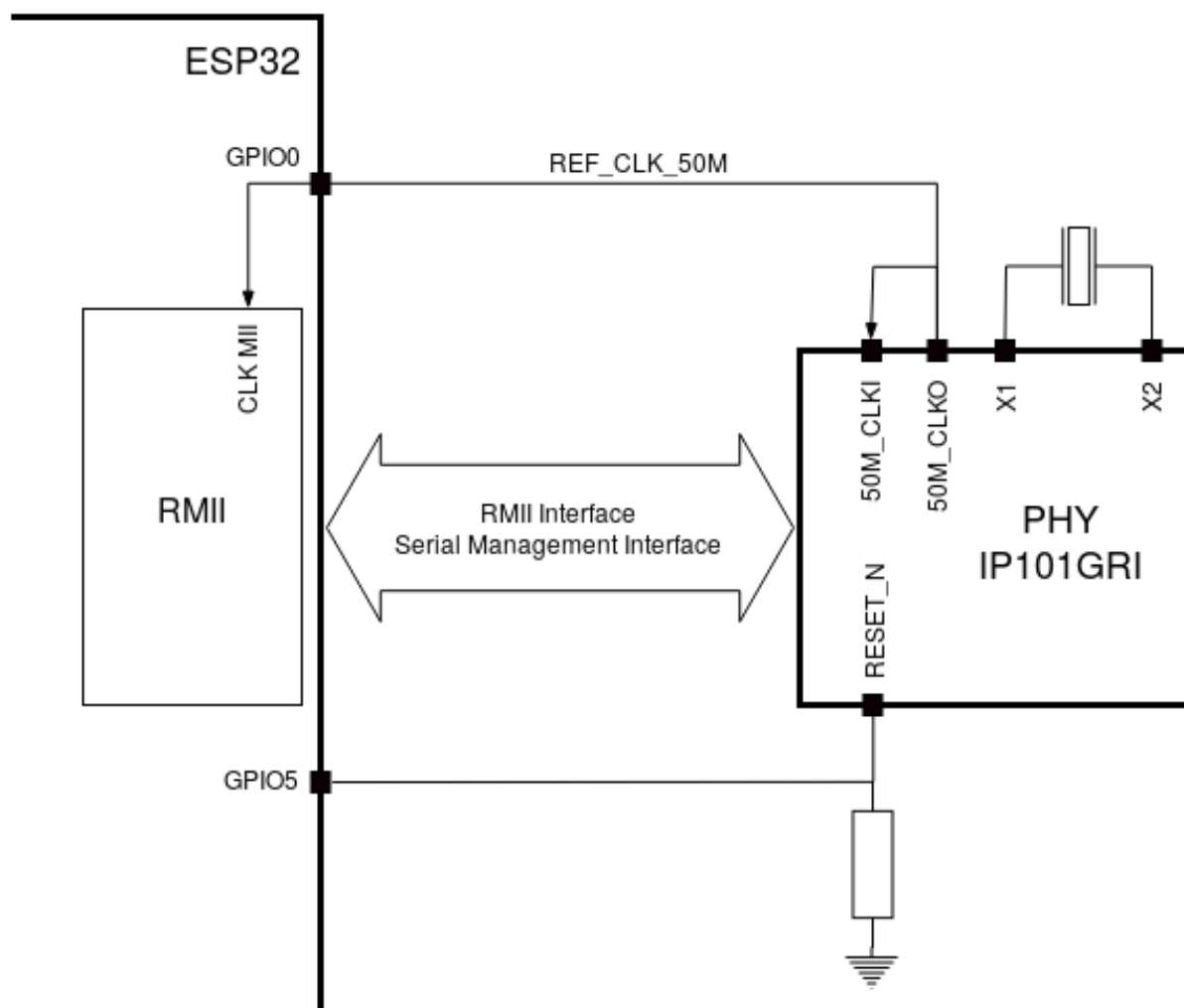


Fig. 24: RMI Clock from IP101GRI PHY

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

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

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

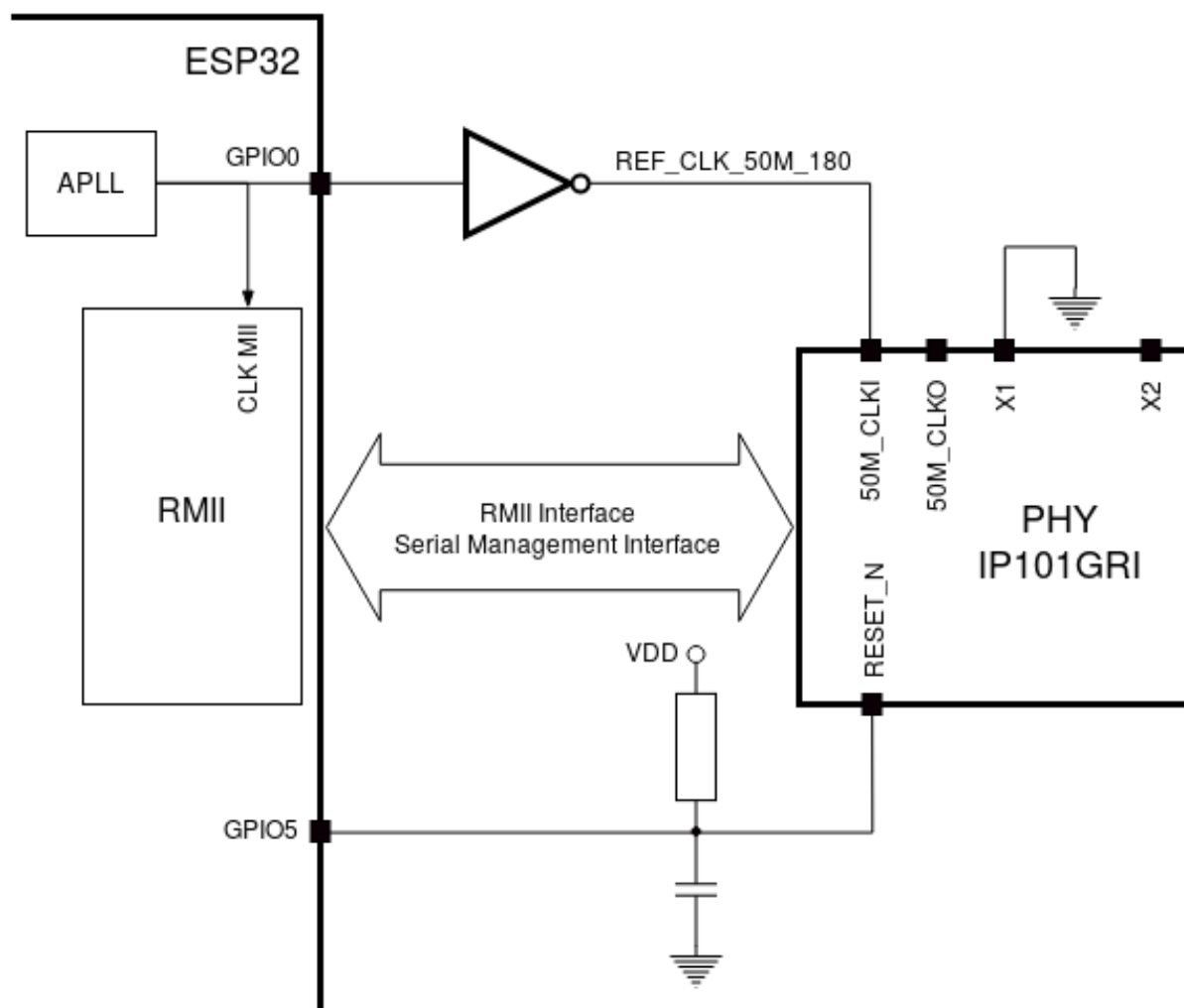


Fig. 25: RMI Clock from ESP Internal APLL

GPIO Allocation

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

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

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

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

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

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

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

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

Note:

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

GPIO Allocation Summary

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

Note:

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

Start Application Development

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

Initial Setup

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

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

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

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

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

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

Other Versions of ESP32-Ethernet-Kit

- [ESP32-Ethernet-Kit V1.0 Getting Started Guide](#)

Related Documents

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

ESP32-Ethernet-Kit V1.0 Getting Started Guide This guide shows how to get started with the ESP32-Ethernet-Kit development board and also provides information about its functionality and configuration options.

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

What You Need

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

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

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

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

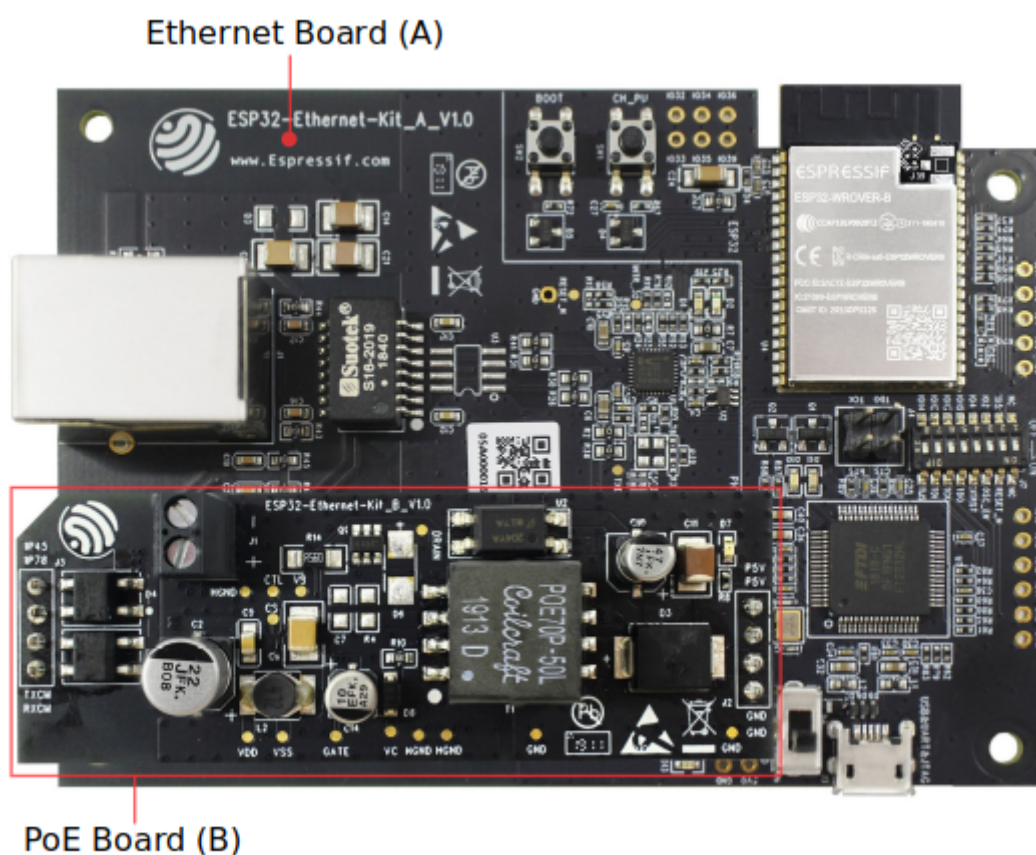


Fig. 26: ESP32-Ethernet-Kit V1.0

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

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

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

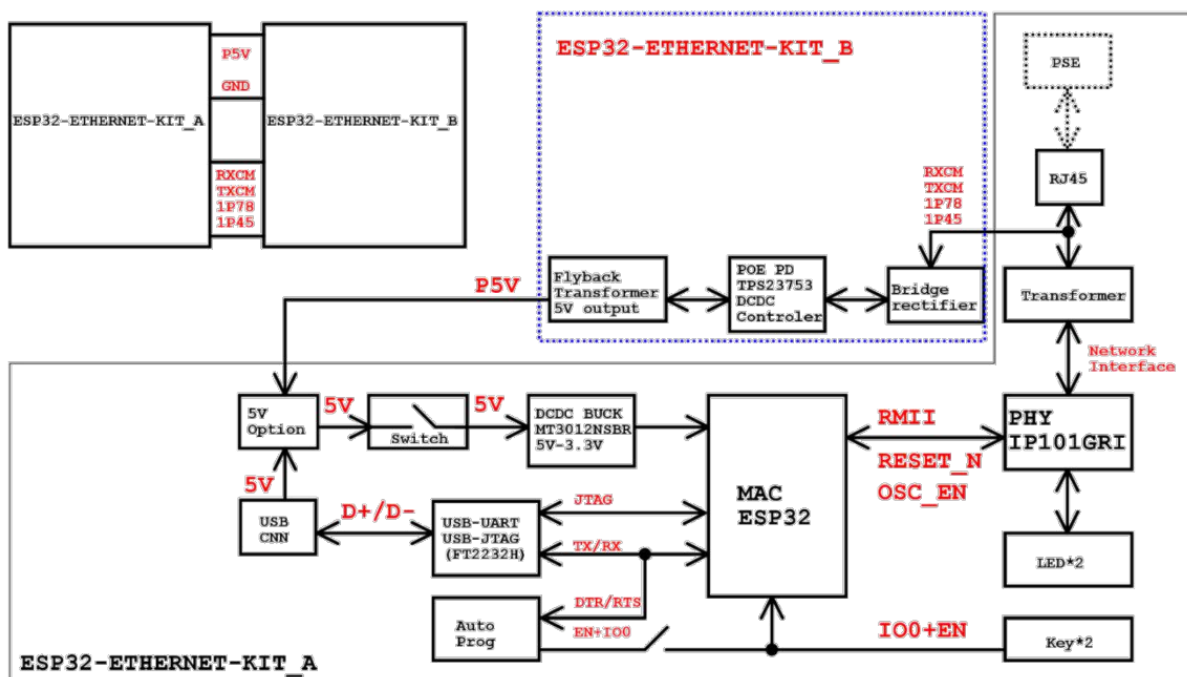


Fig. 27: ESP32-Ethernet-Kit block diagram (click to enlarge)

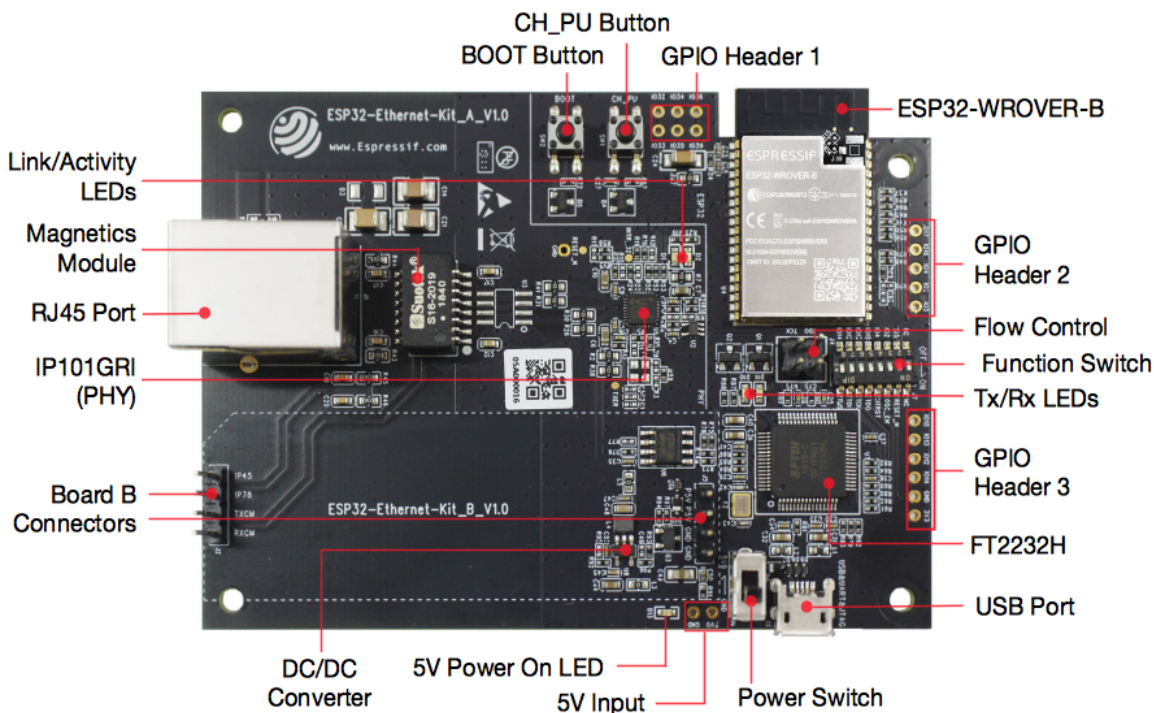


Fig. 28: ESP32-Ethernet-Kit - Ethernet board (A) layout (click to enlarge)

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

| Key Component | Description |
|--------------------|---|
| ESP32-WROVER-B | This ESP32 module features 64-Mbit PSRAM for flexible extended storage and data processing capabilities. |
| GPIO Header 2 | Five unpopulated through-hole solder pads to provide access to selected GPIOs of ESP32. For details, see GPIO Header 2 . |
| Flow Control | A jumper header with access to the board signals. For details, see Flow Control . |
| Function Switch | A DIP switch used to configure the functionality of selected GPIOs of ESP32. For details, see Function Switch . |
| Tx/Rx LEDs | Two LEDs to show the status of UART transmission. |
| GPIO Header 3 | Provides access to some GPIOs of ESP32 that can be used depending on the position of the Function Switch . |
| FT2232HL | The FT2232HL chip serves as a multi-protocol USB-to-serial bridge which can be programmed and controlled via USB to provide communication with ESP32. FT2232HL also features USB-to-JTAG interface which is available on channel A of the chip, while USB-to-serial is on channel B. The FT2232HL chip enhances user-friendliness in terms of application development and debugging. See ESP32-Ethernet-Kit V1.0 Ethernet board (A) schematic . |
| USB Port | USB interface. Power supply for the board as well as the communication interface between a computer and the board. |
| Power Switch | Power On/Off Switch. Toggling toward the Boot button powers the board on, toggling away from Boot powers the board off. |
| 5V Input | The 5V power supply interface can be more convenient when the board is operating autonomously (not connected to a computer). |
| 5V Power On LED | This red LED turns on when power is supplied to the board, either from USB or 5V Input. |
| DC/DC Converter | Provided DC 5 V to 3.3 V conversion, output current up to 2A. |
| Board B Connectors | A pair male header pins for mounting the PoE board (B) . |
| IP101GR | The physical layer (PHY) connection to the Ethernet cable is implemented using the IP101GR chip. The connection between PHY and ESP32 is done through the reduced media-independent interface (RMII), a variant of the media-independent interface (MII) standard. The PHY supports the IEEE 802.3 / 802.3u standard of 10/100Mbps. |
| RJ45 Port | Ethernet network data transmission port. |
| Magnetics Module | The Magnetics are part of the Ethernet specification to protect against faults and transients, including rejection of common mode signals between the transceiver IC and the cable. The magnetics also provide galvanic isolation between the transceiver and the Ethernet device. |
| Link/Activity LEDs | Two LEDs (green and red) that respectively indicate the “Link” and “Activity” statuses of the PHY. |
| BOOT Button | Download button. Holding down BOOT and then pressing CH_PU initiates Firmware Download mode for downloading firmware through the serial port. |
| CH_PU Button | Reset button. |

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

The PoE board (B) has the following features:

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

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



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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Note:

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

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

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

GPIO Allocation Summary

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

Note:

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

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

Initial Setup

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

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

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

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

Related Documents

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

1.4 Installation Step by Step

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

1.4.1 Setting up Development Environment

- *Step 1. Install prerequisites for Windows, Linux or macOS*
- *Step 2. Get ESP-IDF*
- *Step 3. Set up the tools*
- *Step 4. Set up the environment variables*

1.4.2 Creating Your First Project

- *Step 5. Start a Project*
- *Step 6. Connect Your Device*
- *Step 7. Configure*
- *Step 8. Build the Project*
- *Step 9. Flash onto the Device*
- *Step 10. Monitor*

1.5 Step 1. Install prerequisites

Some tools need to be installed on the computer before proceeding to the next steps. Follow the links below for the instructions for your OS:

1.5.1 Standard Setup of Toolchain for Windows

Introduction

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

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

Note: Limitation: the installation path of Python or ESP-IDF must not contain white spaces or parentheses.

Limitation: the installation path of Python or ESP-IDF should not contain special characters (non-ASCII) unless the operating system is configured with "Unicode UTF-8" support. System Administrator can enable the support via Control Panel - Change date, time, or number formats - Administrative tab - Change system locale - check the option "Beta: Use Unicode UTF-8 for worldwide language support" - Ok and reboot the computer.

ESP-IDF Tools Installer

The easiest way to install ESP-IDF's prerequisites is to download one of ESP-IDF Tools Installers from this URL: <https://dl.espressif.com/dl/esp-idf/?idf=4.1>

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

Offline Installer does not require any network connection. The installer contains all required dependencies including [Git For Windows](#).

Components of the installation The installer deploys the following components:

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

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

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

Run ESP-IDF PowerShell Environment:

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

Using the Command Prompt

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

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

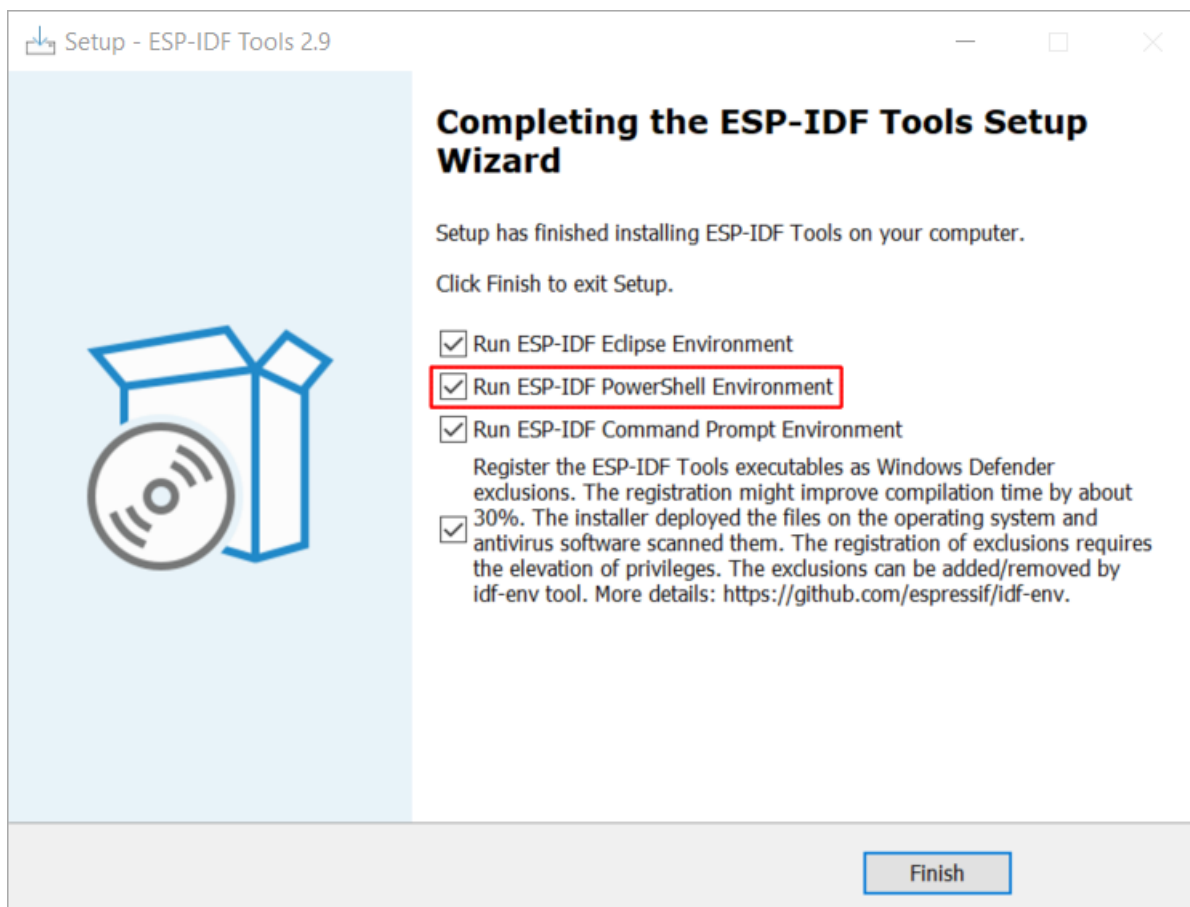


Fig. 30: Completing the ESP-IDF Tools Setup Wizard with Run ESP-IDF PowerShell Environment

```

ESP-IDF PowerShell
Using Python in C:\Users\developer\.espressif\python_env\idf4.1_py3.8_env\Scripts
Python 3.8.7
Using Git in C:\Program Files\Git\cmd/
git version 2.29.2.windows.1
Setting IDF_PATH: C:\Users\developer\Desktop\esp-idf
Adding ESP-IDF tools to PATH...
C:\Users\developer\.espressif\tools\xtensa-esp32-elf\esp-2020r3-8.4.0\xtensa-esp32-elf\bin
C:\Users\developer\.espressif\tools\xtensa-esp32s2-elf\esp-2020r3-8.4.0\xtensa-esp32s2-elf\bin
C:\Users\developer\.espressif\tools\esp32ulp-elf\2.28.51-esp-20191205\esp32ulp-elf-binutils\bin
C:\Users\developer\.espressif\tools\esp32s2ulp-elf\2.28.51-esp-20191205\esp32s2ulp-elf-binutils\bin
C:\Users\developer\.espressif\tools\cmake\3.13.4\bin
C:\Users\developer\.espressif\tools\openocd-esp32\v0.10.0-esp32-20200709\openocd-esp32\bin
C:\Users\developer\.espressif\tools\ninja\1.9.0\
C:\Users\developer\.espressif\tools\idf-exe\1.0.1\
C:\Users\developer\.espressif\tools\ccache\3.7\
C:\Users\developer\Desktop\esp-idf\tools
Checking if Python packages are up to date...
Python requirements from C:\Users\developer\Desktop\esp-idf\requirements.txt are satisfied.

Done! You can now compile ESP-IDF projects.
Go to the project directory and run:
  idf.py build

PS C:\Users\developer\Desktop\esp-idf>

```

Fig. 31: ESP-IDF PowerShell

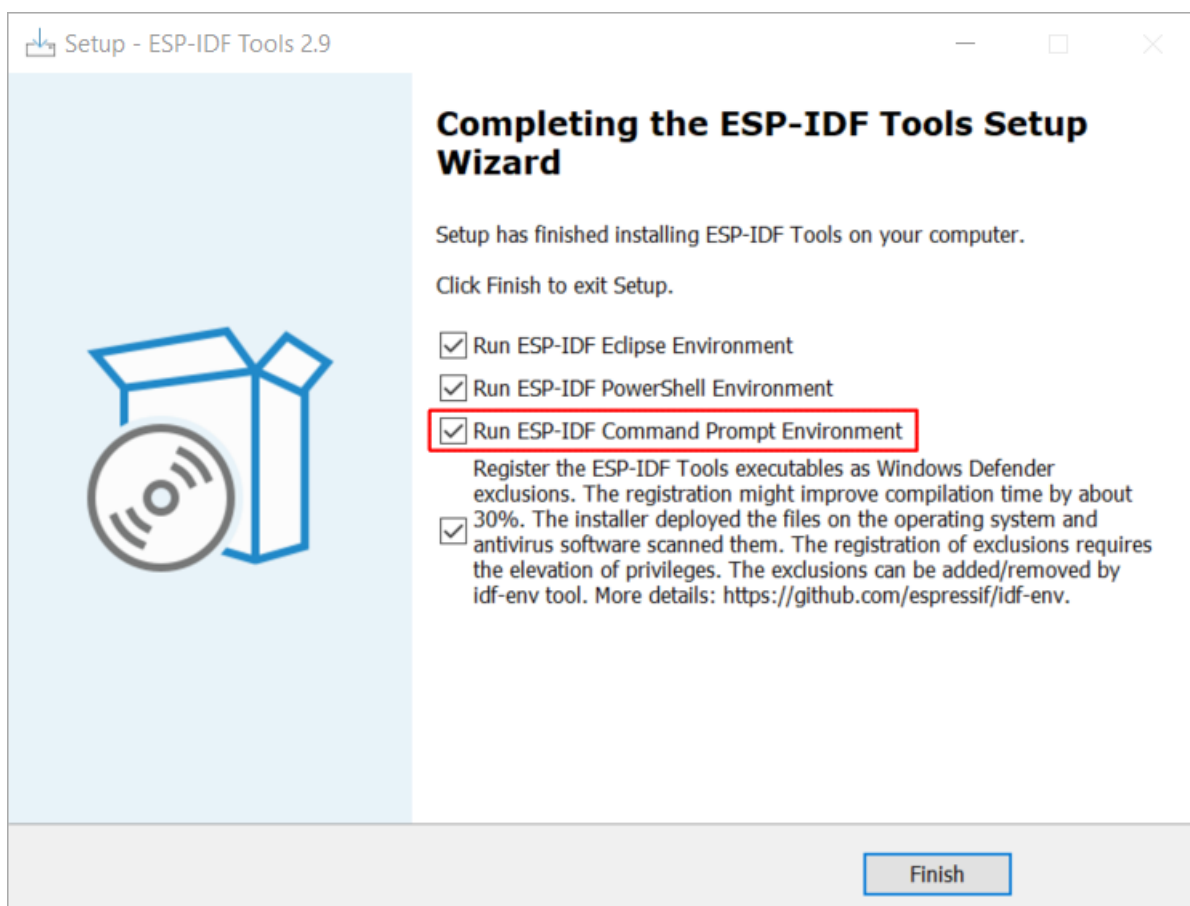
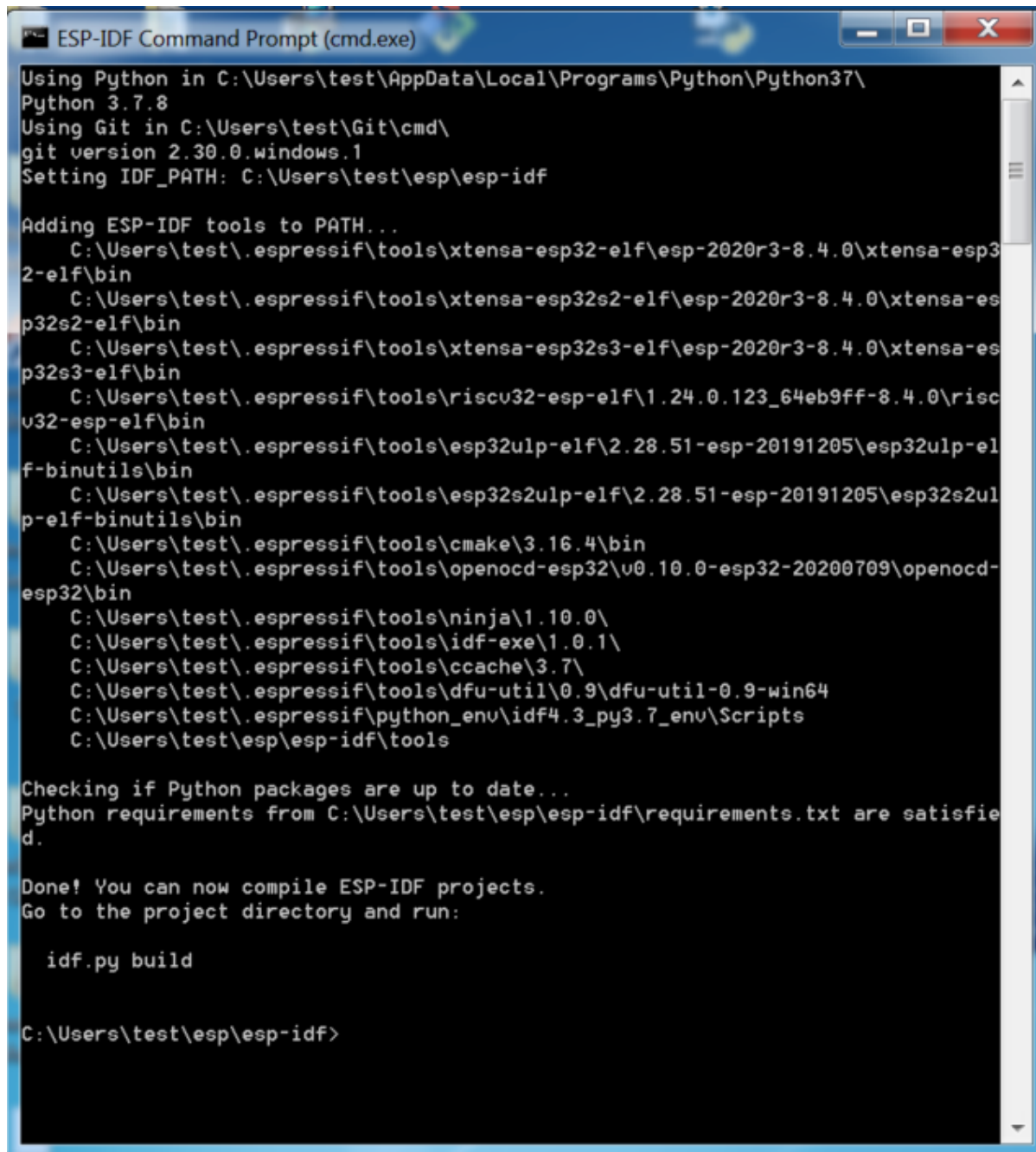


Fig. 32: Completing the ESP-IDF Tools Setup Wizard with Run ESP-IDF Command Prompt (cmd.exe)



```
ESP-IDF Command Prompt (cmd.exe)
Using Python in C:\Users\test\AppData\Local\Programs\Python\Python37\
Python 3.7.8
Using Git in C:\Users\test\Git\cmd\
git version 2.30.0.windows.1
Setting IDF_PATH: C:\Users\test\esp\esp-idf

Adding ESP-IDF tools to PATH...
  C:\Users\test\.espressif\tools\xtensa-esp32-elf\esp-2020r3-8.4.0\xtensa-esp32-elf\bin
  C:\Users\test\.espressif\tools\xtensa-esp32s2-elf\esp-2020r3-8.4.0\xtensa-esp32s2-elf\bin
  C:\Users\test\.espressif\tools\xtensa-esp32s3-elf\esp-2020r3-8.4.0\xtensa-esp32s3-elf\bin
  C:\Users\test\.espressif\tools\riscv32-esp-elf\1.24.0.123_64eb9ff-8.4.0\riscv32-esp-elf\bin
  C:\Users\test\.espressif\tools\esp32ulp-elf\2.28.51-esp-20191205\esp32ulp-elf-binutils\bin
  C:\Users\test\.espressif\tools\esp32s2ulp-elf\2.28.51-esp-20191205\esp32s2ulp-elf-binutils\bin
  C:\Users\test\.espressif\tools\cmake\3.16.4\bin
  C:\Users\test\.espressif\tools\openocd-esp32\v0.10.0-esp32-20200709\openocd-esp32\bin
  C:\Users\test\.espressif\tools\ninja\1.10.0\
  C:\Users\test\.espressif\tools\idf-exe\1.0.1\
  C:\Users\test\.espressif\tools\ccache\3.7\
  C:\Users\test\.espressif\tools\dfu-util\0.9\dfu-util-0.9-win64
  C:\Users\test\.espressif\python_env\idf4.3_py3.7_env\Scripts
  C:\Users\test\esp\esp-idf\tools

Checking if Python packages are up to date...
Python requirements from C:\Users\test\esp\esp-idf\requirements.txt are satisfied.

Done! You can now compile ESP-IDF projects.
Go to the project directory and run:

  idf.py build

C:\Users\test\esp\esp-idf>
```

Fig. 33: ESP-IDF Command Prompt

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

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

Next Steps

If the ESP-IDF Tools Installer has finished successfully, then the development environment setup is complete. Proceed directly to [Step 5. Start a Project](#).

Related Documents

For advanced users who want to customize the install process:

Updating ESP-IDF tools on Windows

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

```
install.bat
```

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

```
install.ps1
```

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

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

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

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

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

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

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

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

1.5.2 Standard Setup of Toolchain for Linux

Install Prerequisites

To compile with ESP-IDF you need to get the following packages:

- CentOS 7:

```
sudo yum -y update && sudo yum install git wget flex bison gperf python3_
↳python3-pip python3-setuptools cmake ninja-build ccache
```

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

- Ubuntu and Debian:

```
sudo apt-get install git wget flex bison gperf python3 python3-pip python3-
↳setuptools cmake ninja-build ccache libffi-dev libssl-dev dfu-util
```

- Arch:

```
sudo pacman -S --needed gcc git make flex bison gperf python-pip cmake ninja_
↳ccache
```

Note: CMake version 3.5 or newer is required for use with ESP-IDF. Older Linux distributions may require updating, enabling of a “backports” repository, or installing of a “cmake3” package rather than “cmake” .

Additional Tips

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

Fixing broken pip on Ubuntu 16.04

Package python3-pip could be broken without possibility to upgrade it. Package has to be removed and installed manually using script [get-pip.py](#):

```
apt remove python3-pip python3-virtualenv; rm -r ~/.local
rm -r ~/.espressif/python_env && python get-pip.py
```

Python 2 deprecation

Python 2 reached its [end of life](#) and support for it in ESP-IDF will be removed soon. Please install Python 3.6 or higher. Instructions for popular Linux distributions are listed above.

Next Steps

To carry on with development environment setup, proceed to [Step 2. Get ESP-IDF](#).

1.5.3 Standard Setup of Toolchain for Mac OS

Install Prerequisites

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

- install pip:

```
sudo easy_install pip
```

- install CMake & Ninja build:

- If you have [HomeBrew](#), you can run:

```
brew install cmake ninja
```

- If you have [MacPorts](#), you can run:

```
sudo port install cmake ninja
```

- Otherwise, consult the [CMake](#) and [Ninja](#) home pages for macOS installation downloads.

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

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

```
xcrun: error: invalid active developer path (/Library/Developer/CommandLineTools), -  
↳missing xcrun at: /Library/Developer/CommandLineTools/usr/bin/xcrun
```

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

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

```
python --version
```

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

```
python3 --version
```

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

Below is an overview of steps to install Python 3.

- Installing with [HomeBrew](#) can be done as follows:

```
brew install python3
```

- If you have [MacPorts](#), you can run:

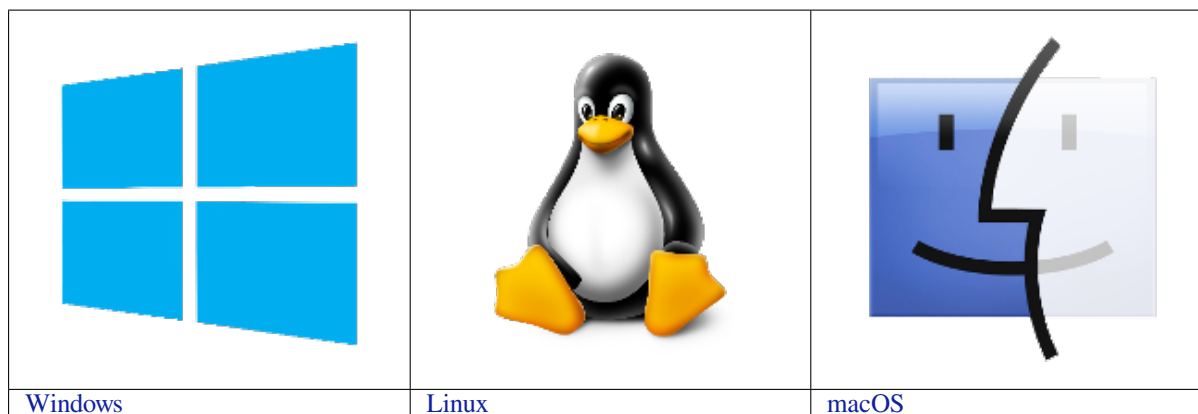
```
sudo port install python38
```

Python 2 deprecation

Python 2 reached its [end of life](#) and support for it in ESP-IDF will be removed soon. Please install Python 3.6 or higher. Instructions for macOS are listed above.

Next Steps

To carry on with development environment setup, proceed to [Step 2. Get ESP-IDF](#).



1.6 Step 2. Get ESP-IDF

To build applications for the ESP32, you need the software libraries provided by Espressif in [ESP-IDF repository](#).

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

Note: This guide uses the directory `~/esp` on Linux and macOS or `%userprofile%\esp` on Windows as an installation folder for ESP-IDF. You can use any directory, but you will need to adjust paths for the commands respectively. Keep in mind that ESP-IDF does not support spaces in paths.

1.6.1 Linux and macOS

Open Terminal, and run the following commands:

```
cd ~/esp
git clone -b v4.1.3 --recursive https://github.com/espressif/esp-idf.git
```

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

Consult [ESP-IDF Versions](#) for information about which ESP-IDF version to use in a given situation.

1.6.2 Windows

In addition to installing the tools, [ESP-IDF Tools Installer](#) for Windows introduced in Step 1 can also download a copy of ESP-IDF.

Consult [ESP-IDF Versions](#) for information about which ESP-IDF version to use in a given situation.

If you wish to download ESP-IDF without the help of ESP-IDF Tools Installer, refer to these [instructions](#).

1.7 Step 3. Set up the tools

Aside from the ESP-IDF, you also need to install the tools used by ESP-IDF, such as the compiler, debugger, Python packages, etc.

1.7.1 Windows

ESP-IDF Tools Installer for Windows introduced in Step 1 installs all the required tools.

If you want to install the tools without the help of ESP-IDF Tools Installer, open the Command Prompt and follow these steps:

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

or with Windows PowerShell

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

1.7.2 Linux and macOS

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

1.7.3 Alternative File Downloads

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

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

Windows

To prefer the Espressif download server when running the ESP-IDF Tools Installer or installing tools from the command line, open the System control panel, then click on Advanced Settings. Add a new Environment Variable (of type either User or System) with the name `IDF_GITHUB_ASSETS` and value `dl.espressif.com/github_assets`. Click OK once done.

If the command line window or ESP-IDF Tools Installer window was already open before you added the new environment variable, you will need to close and reopen it.

While this environment variable is still set, the ESP-IDF Tools Installer and the command line installer will prefer the Espressif download server.

Linux and macOS

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

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

1.7.4 Customizing the tools installation path

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

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

1.8 Step 4. Set up the environment variables

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

1.8.1 Windows

ESP-IDF Tools Installer for Windows creates an “ESP-IDF Command Prompt” shortcut in the Start Menu. This shortcut opens the Command Prompt and sets up all the required environment variables. You can open this shortcut and proceed to the next step.

Alternatively, if you want to use ESP-IDF in an existing Command Prompt window, you can run:

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

or with Windows PowerShell

```
.$HOME/esp/esp-idf/export.ps1
```

1.8.2 Linux and macOS

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

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

Note the space between the leading dot and the path!

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

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

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

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

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

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

1.9 Step 5. Start a Project

Now you are ready to prepare your application for ESP32. You can start with [get-started/hello_world](#) project from [examples](#) directory in IDF.

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

1.9.1 Linux and macOS

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

1.9.2 Windows

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

There is a range of example projects in the `examples` directory in ESP-IDF. You can copy any project in the same way as presented above and run it.

It is also possible to build examples in-place, without copying them first.

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

1.10 Step 6. Connect Your Device

Now connect your ESP32 board to the computer and check under what serial port the board is visible.

Serial ports have the following patterns in their names:

- **Windows:** names like COM1
- **Linux:** starting with `/dev/tty`
- **macOS:** starting with `/dev/cu`.

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

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

1.11 Step 7. Configure

Navigate to your `hello_world` directory from [Step 5. Start a Project](#) and run the project configuration utility `menuconfig`.

1.11.1 Linux and macOS

```
cd ~/esp/hello_world
idf.py menuconfig
```

1.11.2 Windows

```
cd %userprofile%\esp\hello_world
idf.py menuconfig
```

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

```

Espressif IoT Development Framework Configuration
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highlighted
letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc>
to exit, <?> for Help, </> for Search. Legend: [*] built-in [ ] excluded <M> module < > module
capable

  SDK tool configuration --->
  Bootloader config --->
  Security features --->
  Serial flasher config --->
  Partition Table --->
  Optimization level (Debug) --->
  Component config --->

<Select> < Exit > < Help > < Save > < Load >

```

Fig. 34: Project configuration - Home window

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

To navigate and use `menuconfig`, press the following keys:

- Arrow keys for navigation
- Enter to go into a submenu
- Esc to go up one level or exit
- ? to see a help screen for menu or configuration item. Enter key exits the help screen
- Space, or Y and N keys to enable (Yes) and disable (No) configuration items with checkboxes “[*]”
- / to find configuration items

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

1.12 Step 8. Build the Project

Build the project by running:

```
idf.py build
```

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

```

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

... (more lines of build system output)

[527/527] Generating hello-world.bin

```

(continues on next page)

(continued from previous page)

```
esptool.py v2.3.1

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

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

1.13 Step 9. Flash onto the Device

Flash the binaries that you just built onto your ESP32 board by running:

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

Replace `PORT` with your ESP32 board's serial port name from [Step 6. Connect Your Device](#).

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

For more information on `idf.py` arguments, see [idf.py](#).

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

```
Running esptool.py in directory [...]/esp/hello_world
Executing "python [...]/esp-idf/components/esptool_py/esptool/esptool.py -b 460800.
↳write_flash @flash_project_args"...
esptool.py -b 460800 write_flash --flash_mode dio --flash_size detect --flash_freq.
↳40m 0x1000 bootloader/bootloader.bin 0x8000 partition_table/partition-table.
↳0x10000 hello-world.bin
esptool.py v2.3.1
Connecting....
Detecting chip type... ESP32
Chip is ESP32D0WDQ6 (revision 1)
Features: WiFi, BT, Dual Core
Uploading stub...
Running stub...
Stub running...
Changing baud rate to 460800
Changed.
Configuring flash size...
Auto-detected Flash size: 4MB
Flash params set to 0x0220
Compressed 22992 bytes to 13019...
Wrote 22992 bytes (13019 compressed) at 0x00001000 in 0.3 seconds (effective 558.9.
↳kbit/s)...
Hash of data verified.
Compressed 3072 bytes to 82...
Wrote 3072 bytes (82 compressed) at 0x00008000 in 0.0 seconds (effective 5789.3.
↳kbit/s)...
Hash of data verified.
Compressed 136672 bytes to 67544...
Wrote 136672 bytes (67544 compressed) at 0x00010000 in 1.9 seconds (effective 567.
↳5 kbit/s)...
Hash of data verified.
```

(continues on next page)

(continued from previous page)

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

If there are no issues by the end of the flash process, the module will be reset and the “hello_world” application will be running.

1.14 Step 10. Monitor

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

This command launches the *IDF Monitor* application:

```
$ idf.py -p /dev/ttyUSB0 monitor
Running idf_monitor in directory [...]esp/hello_world/build
Executing "python [...]esp-idf/tools/idf_monitor.py -b 115200 [...]esp/hello_
↵world/build/hello-world.elf"...
--- idf_monitor on /dev/ttyUSB0 115200 ---
--- Quit: Ctrl+] | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H ---
ets Jun  8 2016 00:22:57

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

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

```
...
Hello world!
Restarting in 10 seconds...
I (211) cpu_start: Starting scheduler on APP CPU.
Restarting in 9 seconds...
Restarting in 8 seconds...
Restarting in 7 seconds...
```

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

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

```
e000)(Xn@0y.!00(0PW+)00Hn9a~/90!0t500P0~0k00e0ea050jA
~zY00Y(10,1 00 e000)(Xn@0y.!Dr0zY(0 jpi0|0+z5Ymvp
```

If you have such a problem, do the following:

1. Exit the monitor.
2. Go back to *menuconfig*.
3. Go to Component config → ESP32-specific → Main XTAL frequency, then change *CONFIG_ESP32_XTAL_FREQ_SEL* to 26MHz.
4. After that, *build and flash* the application again.

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

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

See also:

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

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

Now you are ready to try some other [examples](#), or go straight to developing your own applications.

1.15 Updating ESP-IDF

You should update ESP-IDF from time to time, as newer versions fix bugs and provide new features. The simplest way to do the update is to delete the existing `esp-idf` folder and clone it again, as if performing the initial installation described in [Step 2. Get ESP-IDF](#).

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

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

Once the new tools are installed, update the environment using the Export script. See instructions at [Step 4. Set up the environment variables](#).

1.16 Related Documents

1.16.1 Establish Serial Connection with ESP32

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

Connect ESP32 to PC

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

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

- CP210x: [CP210x USB to UART Bridge VCP Drivers](#)
- FTDI: [FTDI Virtual COM Port Drivers](#)

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

Check port on Windows

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

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

Check port on Linux and MacOS

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

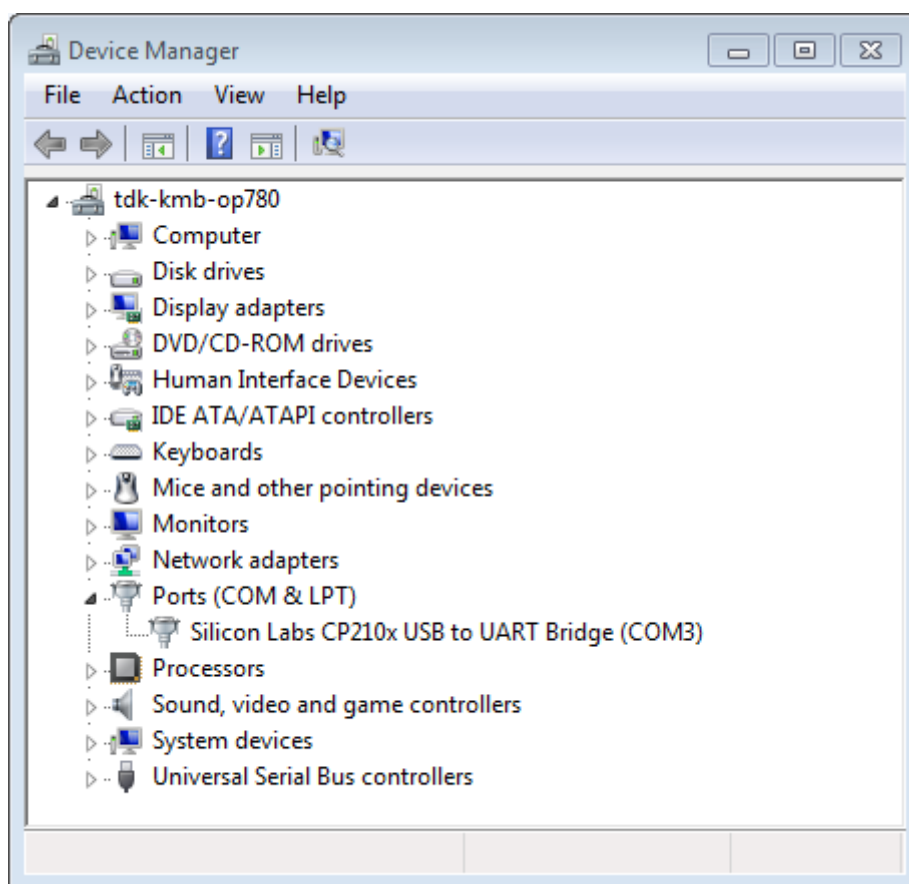


Fig. 35: USB to UART bridge of ESP32-DevKitC in Windows Device Manager

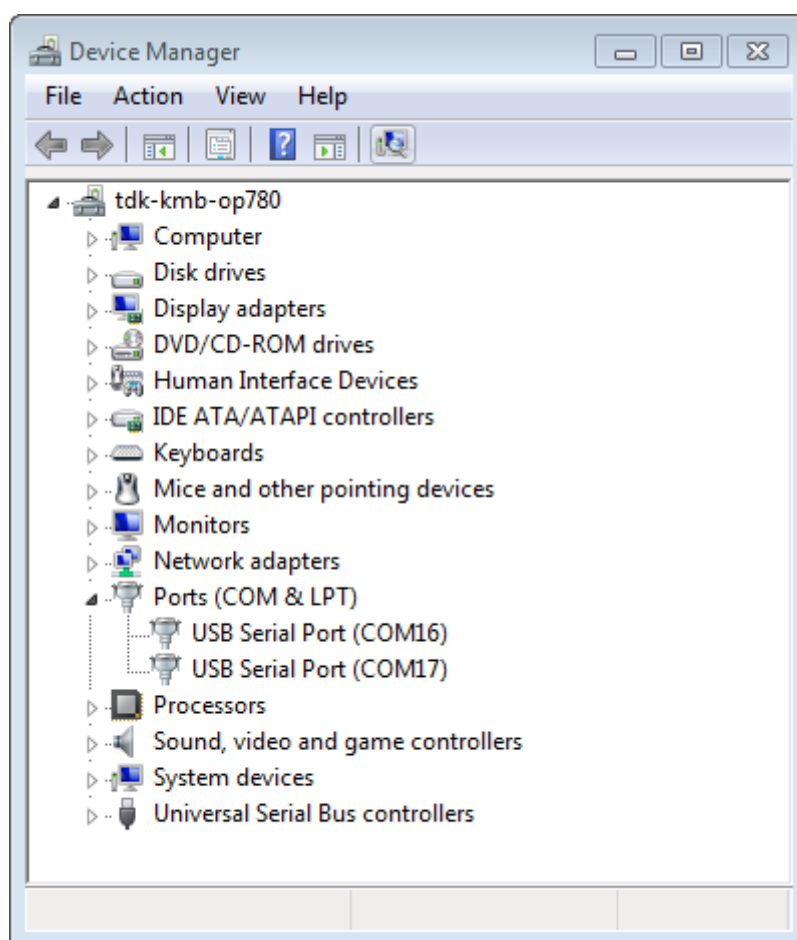


Fig. 36: Two USB Serial Ports of ESP-WROVER-KIT in Windows Device Manager

Linux

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

MacOS

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

Note: MacOS users: if you don't see the serial port then check you have the USB/serial drivers installed as shown in the Getting Started guide for your particular development board. For MacOS High Sierra (10.13), you may also have to explicitly allow the drivers to load. Open System Preferences -> Security & Privacy -> General and check if there is a message shown here about "System Software from developer ..." where the developer name is Silicon Labs or FTDI.

Adding user to dialout on Linux

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

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

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

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

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

Verify serial connection

Now verify that the serial connection is operational. You can do this using a serial terminal program. In this example we will use [PuTTY SSH Client](#) that is available for both Windows and Linux. You can use other serial program and set communication parameters like below.

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

Then open serial port in terminal and check, if you see any log printed out by ESP32. The log contents will depend on application loaded to ESP32. An example log by ESP32 is shown below.

```
ets Jun  8 2016 00:22:57
rst:0x5 (DEEPSLEEP_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
ets Jun  8 2016 00:22:57
rst:0x7 (TGWDT_SYS_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0x00
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0008,len:8
load:0x3fff0010,len:3464
load:0x40078000,len:7828
load:0x40080000,len:252
entry 0x40080034
I (44) boot: ESP-IDF v2.0-rc1-401-gf9fba35 2nd stage bootloader
I (45) boot: compile time 18:48:10
...
```

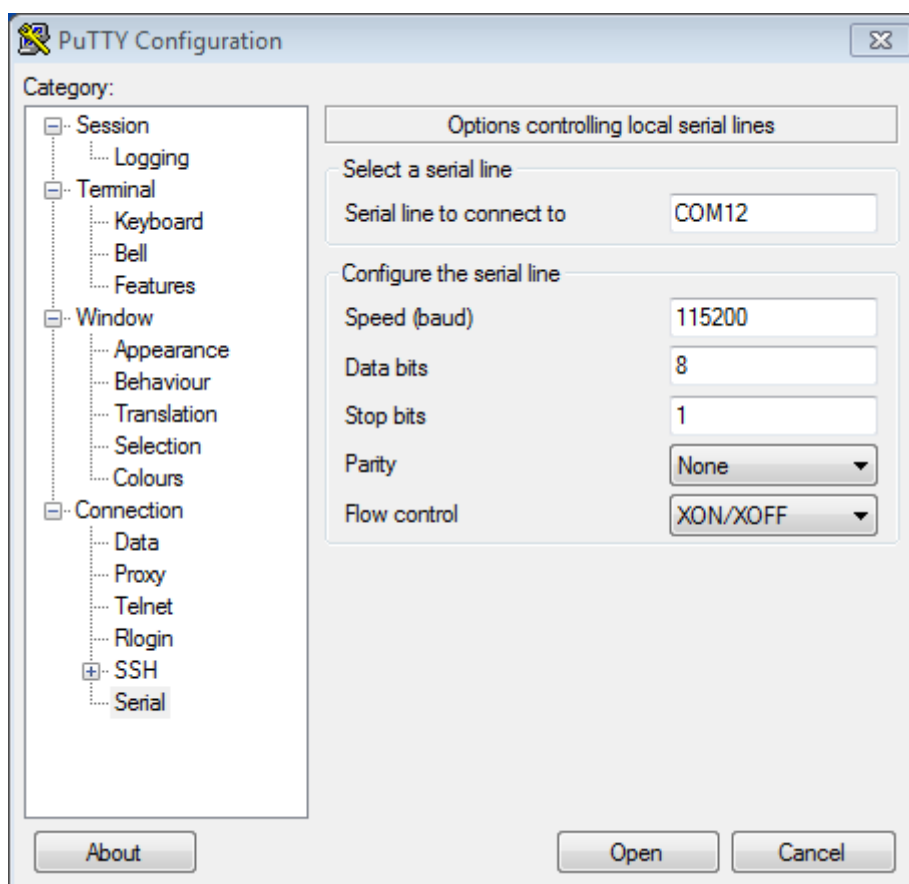


Fig. 37: Setting Serial Communication in PuTTY on Windows

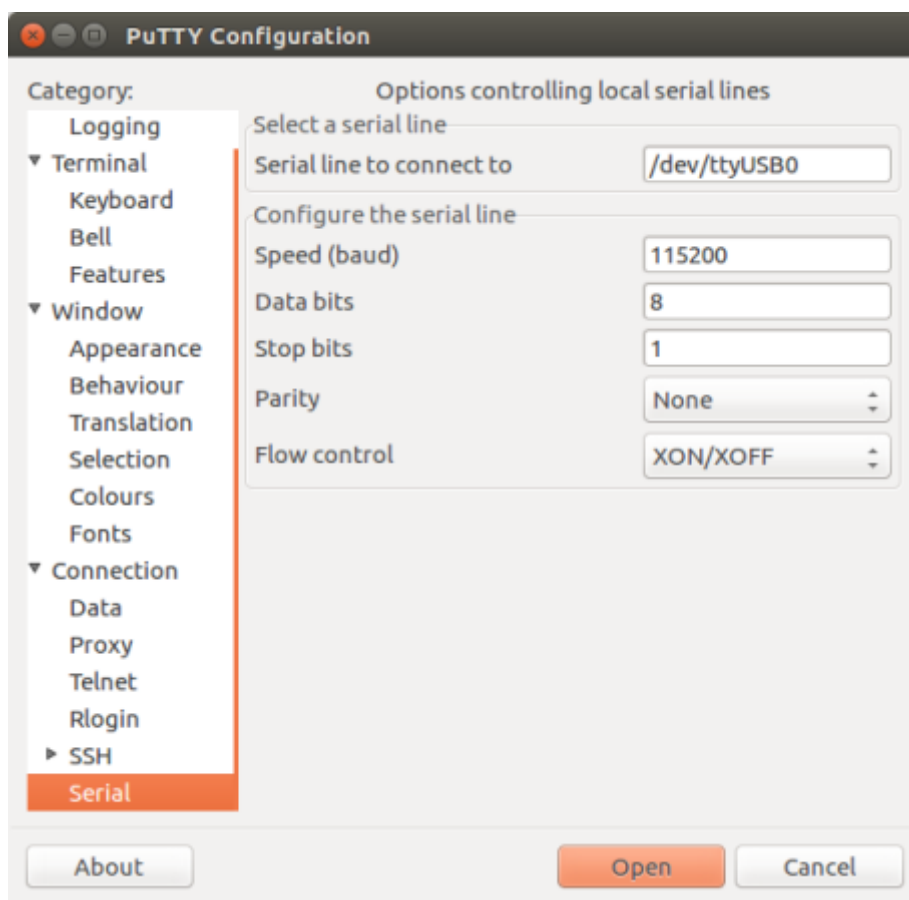


Fig. 38: Setting Serial Communication in PuTTY on Linux

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

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

Note: Close serial terminal after verification that communication is working. In the next step we are going to use a different application to upload a new firmware to ESP32. This application will not be able to access serial port while it is open in terminal.

If you got here from *Step 6. Connect Your Device* when installing s/w for ESP32 development, then you can continue with *Step 7. Configure*.

1.16.2 Build and Flash with Eclipse IDE

ESP-IDF V4.0 has a new CMake-based build system as the default build system.

There is a new ESP-IDF Eclipse Plugin that works with the CMake-based build system. Please refer to <https://github.com/espressif/idf-eclipse-plugin/blob/master/README.md> for further instructions.

If you require Eclipse IDE support for legacy ESP_IDF Make build system, you can follow the *legacy GNU Make build system Getting Started guide* which has steps for *Building and Flashing with Eclipse IDE*.

1.16.3 IDF Monitor

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

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

For the legacy GNU Make system, run `make monitor`.

Keyboard Shortcuts

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

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

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

IDF-specific features

Automatic Address Decoding Whenever ESP-IDF outputs a hexadecimal code address of the form 0x4_____, IDF Monitor uses [addr2line](#) to look up the location in the source code and find the function name.

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

```
Guru Meditation Error of type StoreProhibited occurred on core 0. Exception was
↳unhandled.
Register dump:
PC      : 0x400f360d  PS      : 0x00060330  A0      : 0x800dbf56  A1      :
↳0x3ffb7e00
A2      : 0x3ffb136c  A3      : 0x00000005  A4      : 0x00000000  A5      :
↳0x00000000
A6      : 0x00000000  A7      : 0x00000080  A8      : 0x00000000  A9      :
↳0x3ffb7dd0
A10     : 0x00000003  A11     : 0x00060f23  A12     : 0x00060f20  A13     :
↳0x3ffba6d0
```

(continues on next page)

(continued from previous page)

```

A14      : 0x00000047  A15      : 0x0000000f  SAR       : 0x00000019  EXCCAUSE:↵
↵0x0000001d
EXCVADDR: 0x00000000  LBEG     : 0x4000c46c  LEND      : 0x4000c477  LCOUNT  :↵
↵0x00000000

Backtrace: 0x400f360d:0x3ffb7e00 0x400dbf56:0x3ffb7e20 0x400dbf5e:0x3ffb7e40↵
↵0x400dbf82:0x3ffb7e60 0x400d071d:0x3ffb7e90

```

IDF Monitor adds more details to the dump:

```

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

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

```

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

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

Launching GDB with GDBStub By default, if esp-idf crashes, the panic handler prints relevant registers and the stack dump (similar to the ones above) over the serial port. Then it resets the board.

Optionally, the panic handler can be configured to run GDBStub, the tool which can communicate with [GDB](#) project debugger. GDBStub allows to read memory, examine call stack frames and variables, etc. It is not as versatile as JTAG debugging, but this method does not require any special hardware.

To enable GDBStub, open the project configuration menu (`idf.py menuconfig`) and set [CONFIG_ESP32_PANIC](#) to Invoke GDBStub.

In this case, if the panic handler is triggered, as soon as IDF Monitor sees that GDBStub has loaded, it automatically pauses serial monitoring and runs GDB with necessary arguments. After GDB exits, the board is reset via the RTS serial line. If this line is not connected, please reset the board manually by pressing its Reset button.

In the background, IDF Monitor runs the following command:

```
xtensa-esp32-elf-gdb -ex "set serial baud BAUD" -ex "target remote PORT" -ex.
↳ interrupt build/PROJECT.elf
```

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

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

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

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

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

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

Examples Of Filtering Rules:

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

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

```
load:0x40078000,len:13564
entry 0x40078d4c
E (31) esp_image: image at 0x30000 has invalid magic byte
W (31) esp_image: image at 0x30000 has invalid SPI mode 255
E (39) boot: Factory app partition is not bootable
I (568) cpu_start: Pro cpu up.
I (569) heap_init: Initializing. RAM available for dynamic allocation:
I (603) cpu_start: Pro cpu start user code
D (309) light_driver: [light_init, 74]:status: 1, mode: 2
```

(continues on next page)

(continued from previous page)

```
D (318) vfs: esp_vfs_register_fd_range is successful for range <54; 64) and VFS ID_
↵1
I (328) wifi: wifi driver task: 3ffdbf84, prio:23, stack:4096, core=0
```

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

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

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

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

Known Issues with IDF Monitor

Issues Observed on Windows

- If in the Windows environment you receive the error “winpty: command not found”, fix it by running `pacman -S winpty`.
- Arrow keys, as well as some other keys, do not work in GDB due to Windows Console limitations.
- Occasionally, when “idf.py” or “make” exits, it might stall for up to 30 seconds before IDF Monitor resumes.
- When “gdb” is run, it might stall for a short time before it begins communicating with the GDBStub.

1.16.4 Customized Setup of Toolchain

Instead of downloading binary toolchain from Espressif website (see [Step 3. Set up the tools](#)) you may build the toolchain yourself.

If you can't think of a reason why you need to build it yourself, then probably it's better to stick with the binary version. However, here are some of the reasons why you might want to compile it from source:

- if you want to customize toolchain build configuration
- if you want to use a different GCC version (such as 4.8.5)
- if you want to hack gcc or newlib or libstdc++
- if you are curious and/or have time to spare
- if you don't trust binaries downloaded from the Internet

In any case, here are the instructions to compile the toolchain yourself.

Setup Windows Toolchain from Scratch

This is a step-by-step alternative to running the [ESP-IDF Tools Installer](#) for the CMake-based build system. Installing all of the tools by hand allows more control over the process, and also provides the information for advanced users to customize the install.

To quickly setup the toolchain and other tools in standard way, using the ESP-IDF Tools installer, proceed to section [Standard Setup of Toolchain for Windows](#).

Note: The GNU Make based build system requires the [MSYS2](#) Unix compatibility environment on Windows. The CMake-based build system does not require this environment.

Get ESP-IDF

Note: Previous versions of ESP-IDF used the **MSYS2 bash terminal** command line. The current cmake-based build system can run in the regular **Windows Command Prompt** which is used here.

If you use a bash-based terminal or PowerShell, please note that some command syntax will be different to what is shown below.

Open Command Prompt and run the following commands:

```
mkdir %userprofile%\esp
cd %userprofile%\esp
git clone -b v4.1.3 --recursive https://github.com/espressif/esp-idf.git
```

ESP-IDF will be downloaded into %userprofile%\esp\esp-idf.

Consult [ESP-IDF Versions](#) for information about which ESP-IDF version to use in a given situation.

Note: The `git clone` option `-b v4.1.3` tells git to clone the tag in the ESP-IDF repository `git clone` corresponding to this version of the documentation.

Note: As a fallback, it is also possible to download a zip file of this stable release from the [Releases page](#). Do not download the “Source code” zip file(s) generated automatically by GitHub, they do not work with ESP-IDF.

Note: Do not miss the `--recursive` option. If you have already cloned ESP-IDF without this option, run another command to get all the submodules:

```
cd esp-idf
git submodule update --init
```

Tools

cmake Download the latest stable release of [CMake](#) for Windows and run the installer.

When the installer asks for Install Options, choose either “Add CMake to the system PATH for all users” or “Add CMake to the system PATH for the current user” .

Ninja build

Note: Ninja currently only provides binaries for 64-bit Windows. It is possible to use CMake and `idf.py` with other build tools, such as mingw-make, on 32-bit windows. However this is currently undocumented.

Download the [ninja](#) latest stable Windows release from the ([download page](#)).

The Ninja for Windows download is a .zip file containing a single `ninja.exe` file which needs to be unzipped to a directory which is then *added to your Path* (or you can choose a directory which is already on your Path).

Python Download the latest [Python](#) for Windows installer, and run it.

The “Customise” step of the Python installer gives a list of options. The last option is “Add python.exe to Path” . Change this option to select “Will be installed” .

Once Python is installed, open a Windows Command Prompt from the Start menu and run the following command:

```
pip install --user pyserial
```

Toolchain Setup Download the precompiled Windows toolchain:

https://dl.espressif.com/dl/xtensa-esp32-elf-gcc8_4_0-esp-2020r3-win32.zip

Unzip the zip file to C:\Program Files (or some other location). The zip file contains a single directory xtensa-esp32-elf.

Next, the bin subdirectory of this directory must be *added to your Path*. For example, the directory to add may be C:\Program Files\xtensa-esp32-elf\bin.

Note: If you already have the MSYS2 environment (for use with the “GNU Make” build system) installed, you can skip the separate download and add the directory C:\msys32\opt\xtensa-esp32-elf\bin to the Path instead, as the toolchain is included in the MSYS2 environment.

Adding Directory to Path To add any new directory to your Windows Path environment variable:

Open the System control panel and navigate to the Environment Variables dialog. (On Windows 10, this is found under Advanced System Settings).

Double-click the Path variable (either User or System Path, depending if you want other users to have this directory on their path.) Go to the end of the value, and append ; <new value>.

Next Steps To carry on with development environment setup, proceed to *Step 3. Set up the tools*.

Setup Linux Toolchain from Scratch

The following instructions are alternative to downloading binary toolchain from Espressif website. To quickly setup the binary toolchain, instead of compiling it yourself, backup and proceed to section *Standard Setup of Toolchain for Linux*.

Note: The reason you might need to build your own toolchain is to solve the Y2K38 problem (time_t expand to 64 bits instead of 32 bits).

Install Prerequisites To compile with ESP-IDF you need to get the following packages:

- CentOS 7:

```
sudo yum -y update && sudo yum install git wget ncurses-devel flex bison gperf_  
↳python3 python3-pip cmake ninja-build ccache
```

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

- Ubuntu and Debian:

```
sudo apt-get install git wget libncurses-dev flex bison gperf python3 python3-_  
↳pip python3-setuptools python3-serial python3-cryptography python3-future_  
↳python3-pyparsing python3-pyelftools cmake ninja-build ccache libffi-dev_  
↳libssl-dev dfu-util
```

- Arch:

```
sudo pacman -Sy --needed gcc git make ncurses flex bison gperf python-pyserial_  
↳python-cryptography python-future python-pyparsing python-pyelftools cmake_  
↳ninja ccache dfu-util
```


Note: CMake version 3.5 or newer is required for use with ESP-IDF. Older Linux distributions may require updating, enabling of a “backports” repository, or installing of a “cmake3” package rather than “cmake” .

Compile the Toolchain from Source

- Install dependencies:

- CentOS 7:

```
sudo yum install gawk gperf grep gettext ncurses-devel python3 python3-  
↳devel automake bison flex texinfo help2man libtool make
```

- Ubuntu pre-16.04:

```
sudo apt-get install gawk gperf grep gettext libncurses-dev python python-  
↳dev automake bison flex texinfo help2man libtool make
```

- Ubuntu 16.04 or newer:

```
sudo apt-get install gawk gperf grep gettext python python-dev automake  
↳bison flex texinfo help2man libtool libtool-bin make
```

- Debian 9:

```
sudo apt-get install gawk gperf grep gettext libncurses-dev python python-  
↳dev automake bison flex texinfo help2man libtool libtool-bin make
```

- Arch:

```
sudo pacman -Sy --needed python-pip
```

Create the working directory and go into it:

```
mkdir -p ~/esp  
cd ~/esp
```

Download crosstool-NG and build it:

```
git clone https://github.com/espressif/crosstool-NG.git  
cd crosstool-NG  
git checkout esp-2020r3  
git submodule update --init  
./bootstrap && ./configure --enable-local && make
```

Note: To create a toolchain with support for 64-bit `time_t`, you need to remove the `--enable-newlib-long-time_t` option from the `crosstool-NG/samples/xtensa-esp32-elf/crosstool.config` file in 33 and 43 lines.

Build the toolchain:

```
./ct-ng xtensa-esp32-elf  
./ct-ng build  
chmod -R u+w builds/xtensa-esp32-elf
```

Toolchain will be built in `~/esp/crosstool-NG/builds/xtensa-esp32-elf`. Follow [instructions for standard setup](#) to add the toolchain to your PATH.

Python 2 deprecation Python 2 reached its [end of life](#) and support for it in ESP-IDF will be removed soon. Please install Python 3.6 or higher. Instructions for popular Linux distributions are listed above.

Next Steps To carry on with development environment setup, proceed to *Step 2. Get ESP-IDF*.

Setup Toolchain for Mac OS from Scratch

Package Manager To set up the toolchain from scratch, rather than *downloading a pre-compiled toolchain*, you will need to install either the [MacPorts](#) or [homebrew](#) package manager.

MacPorts needs a full XCode installation, while homebrew only needs XCode command line tools.

See *Customized Setup of Toolchain* section for some of the reasons why installing the toolchain from scratch may be necessary.

Install Prerequisites

- install pip:

```
sudo easy_install pip
```

- install pyserial:

```
pip install --user pyserial
```

- install CMake & Ninja build:

- If you have HomeBrew, you can run:

```
brew install cmake ninja
```

- If you have MacPorts, you can run:

```
sudo port install cmake ninja
```

Compile the Toolchain from Source

- Install dependencies:

- with MacPorts:

```
sudo port install gsed gawk binutils gperf grep gettext wget libtool_  
↪autoconf automake make
```

- with homebrew:

```
brew install gnu-sed gawk binutils gperftools gettext wget help2man_  
↪libtool autoconf automake make
```

Create a case-sensitive filesystem image:

```
hdiutil create ~/esp/crosstool.dmg -volname "ctng" -size 10g -fs "Case-sensitive_  
↪HFS+"
```

Mount it:

```
hdiutil mount ~/esp/crosstool.dmg
```

Create a symlink to your work directory:

```
mkdir -p ~/esp  
ln -s /Volumes/ctng ~/esp/ctng-volume
```

Go into the newly created directory:

```
cd ~/esp/ctng-volume
```

Download `crosstool-NG` and build it:

```
git clone https://github.com/espressif/crosstool-NG.git
cd crosstool-NG
git checkout esp-2020r3
git submodule update --init
./bootstrap && ./configure --enable-local && make
```

Build the toolchain:

```
./ct-ng xtensa-esp32-elf
./ct-ng build
chmod -R u+w builds/xtensa-esp32-elf
```

Toolchain will be built in `~/esp/ctng-volume/crosstool-NG/builds/xtensa-esp32-elf`. To use it, you need to add `~/esp/ctng-volume/crosstool-NG/builds/xtensa-esp32-elf/bin` to `PATH` environment variable.

Python 2 deprecation Python 2 reached its [end of life](#) and support for it in ESP-IDF will be removed soon. Please install Python 3.6 or higher. Instructions for macOS are listed above.

Next Steps To carry on with development environment setup, proceed to [Step 2. Get ESP-IDF](#).

1.16.5 Get Started (Legacy GNU Make)

Note: Since ESP-IDF V4.0, the default build system is based on CMake. This documentation is for the legacy build system based on GNU Make. Support for this build system may be removed in future major releases.

This document is intended to help you set up the software development environment for the hardware based on Espressif ESP32.

After that, a simple example will show you how to use ESP-IDF (Espressif IoT Development Framework) for menu configuration, then how to build and flash firmware onto an ESP32 board.

Note: This is documentation for stable version v4.1.3 of ESP-IDF. Other [ESP-IDF Versions](#) are also available.

Introduction

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

- Wi-Fi (2.4 GHz band)
- Bluetooth 4.2
- Dual high performance cores
- Ultra Low Power co-processor
- Several peripherals

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

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

What You Need

Hardware:

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

Software:

- **Toolchain** to build the **Application** for ESP32
- **ESP-IDF** that essentially contains API (software libraries and source code) for ESP32 and scripts to operate the **Toolchain**
- **Text editor** to write programs (**Projects**) in C, e.g., [Eclipse](#)

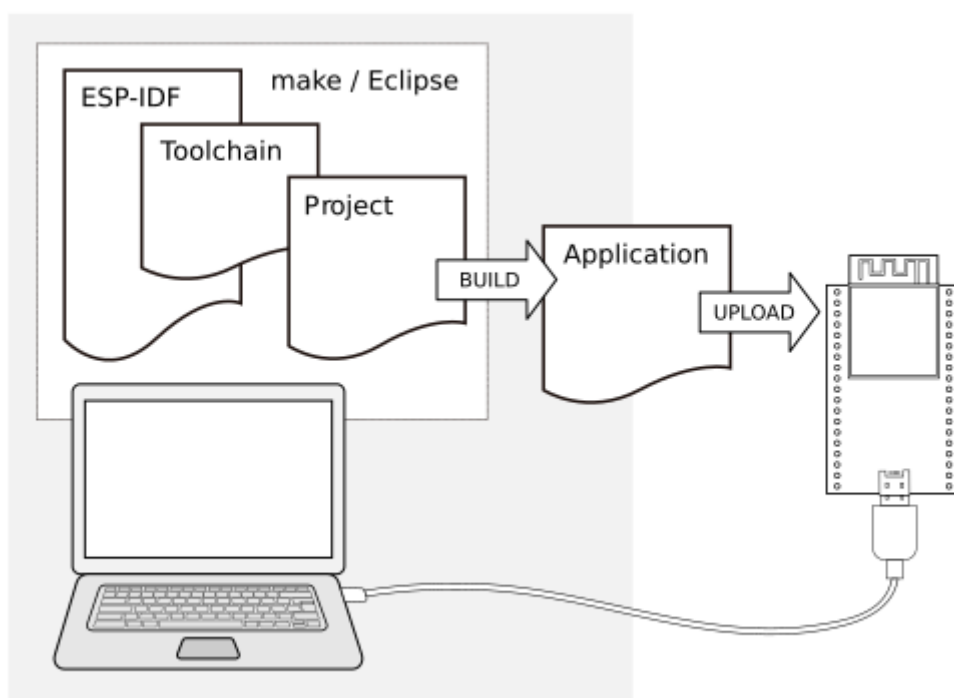


Fig. 39: Development of applications for ESP32

Development Board Overviews

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

Installation Step by Step

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

Setting up Development Environment

- *Step 1. Set up the Toolchain for Windows, Linux or macOS*
- *Step 2. Get ESP-IDF*
- *Step 3. Set Environment Variables*
- *Step 4. Install the Required Python Packages*

Creating Your First Project

- [Step 5. Start a Project](#)
- [Step 6. Connect Your Device](#)
- [Step 7. Configure](#)
- [Step 8. Build and Flash](#)
- [Step 9. Monitor](#)

Step 1. Set up the Toolchain

The toolchain is a set of programs for compiling code and building applications.

The quickest way to start development with ESP32 is by installing a prebuilt toolchain. Pick up your OS below and follow the provided instructions.

Standard Setup of Toolchain for Windows (Legacy GNU Make)

Note: Since ESP-IDF V4.0, the default build system is based on CMake. This documentation is for the legacy build system based on GNU Make. Support for this build system may be removed in future major releases.

Introduction Windows doesn't have a built-in "make" environment, so as well as installing the toolchain you will need a GNU-compatible environment. We use the [MSYS2](#) environment to provide this. You don't need to use this environment all the time (you can use [Eclipse](#) or some other front-end), but it runs behind the scenes.

Toolchain Setup The quick setup is to download the Windows all-in-one toolchain & MSYS2 zip file from [dl.espressif.com](https://dl.espressif.com/dl/esp32_win32_msys2_environment_and_esp2020r2_toolchain-20200601.zip):

https://dl.espressif.com/dl/esp32_win32_msys2_environment_and_esp2020r2_toolchain-20200601.zip

Unzip the zip file to C:\ (or some other location, but this guide assumes C:\) and it will create an `msys32` directory with a pre-prepared environment.

Important: If another toolchain location is used (different than the default `C:\msys32`), please ensure that the path where the all-in-one toolchain gets unzipped is a plain ASCII, contains no spaces, symlinks or accents.

Check it Out Open a MSYS2 MINGW32 terminal window by running `C:\msys32\mingw32.exe`. The environment in this window is a bash shell. Create a directory named `esp` that is a default location to develop ESP32 applications. To do so, run the following shell command:

```
mkdir -p ~/esp
```

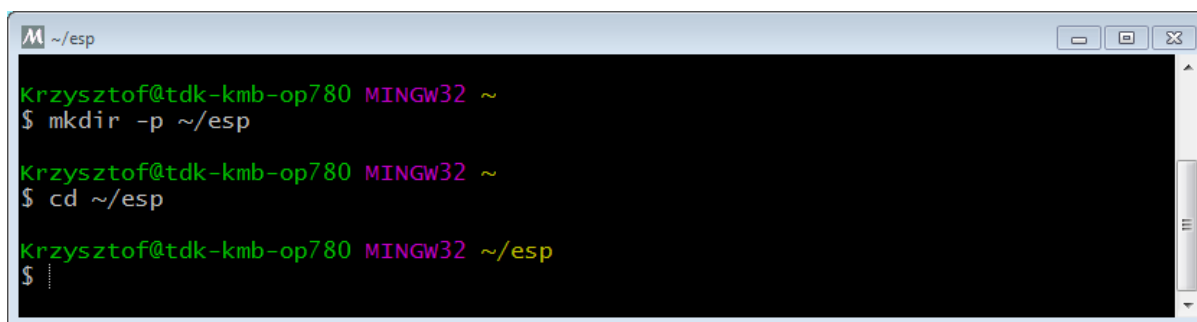
By typing `cd ~/esp` you can then move to the newly created directory. If there are no error messages you are done with this step.

Use this window in the following steps setting up development environment for ESP32.

Next Steps To carry on with development environment setup, proceed to section [Step 2. Get ESP-IDF](#).

Updating The Environment When IDF is updated, sometimes new toolchains are required or new requirements are added to the Windows MSYS2 environment. To move any data from an old version of the precompiled environment to a new one:

- Take the old MSYS2 environment (ie `C:\msys32`) and move/rename it to a different directory (ie `C:\msys32_old`).
- Download the new precompiled environment using the steps above.



```

M ~/esp
Krzysztof@tdk-kmb-op780 MINGW32 ~
$ mkdir -p ~/esp

Krzysztof@tdk-kmb-op780 MINGW32 ~
$ cd ~/esp

Krzysztof@tdk-kmb-op780 MINGW32 ~/esp
$

```

Fig. 40: MSYS2 MINGW32 shell window

- Unzip the new MSYS2 environment to `C:\msys32` (or another location).
- Find the old `C:\msys32_old\home` directory and move this into `C:\msys32`.
- You can now delete the `C:\msys32_old` directory if you no longer need it.

You can have independent different MSYS2 environments on your system, as long as they are in different directories.

There are *also steps to update the existing environment without downloading a new one*, although this is more complex.

Related Documents

Setup Windows Toolchain from Scratch (Legacy GNU Make)

Note: Since ESP-IDF V4.0, the default build system is based on CMake. This documentation is for the legacy build system based on GNU Make. Support for this build system may be removed in future major releases.

Setting up the environment gives you some more control over the process, and also provides the information for advanced users to customize the install. The *pre-built environment*, addressed to less experienced users, has been prepared by following these steps.

To quickly setup the toolchain in standard way, using a prebuilt environment, proceed to section *Standard Setup of Toolchain for Windows (Legacy GNU Make)*.

Configure Toolchain & Environment from Scratch This process involves installing **MSYS2**, then installing the **MSYS2** and Python packages which ESP-IDF uses, and finally downloading and installing the Xtensa toolchain.

- Navigate to the **MSYS2** installer page and download the `msys2-i686-xxxxxxx.exe` installer executable (we only support a 32-bit MSYS environment, it works on both 32-bit and 64-bit Windows.) At time of writing, the latest installer is `msys2-i686-20161025.exe`.
- Run through the installer steps. **Uncheck the “Run MSYS2 32-bit now” checkbox at the end.**
- Once the installer exits, open Start Menu and find “MSYS2 MinGW 32-bit” to run the terminal. (*Why launch this different terminal? MSYS2 has the concept of different kinds of environments. The default “MSYS” environment is Cygwin-like and uses a translation layer for all Windows API calls. We need the “MinGW” environment in order to have a native Python which supports COM ports.*)
- The ESP-IDF repository on github contains a script in the tools directory titled `windows_install_prerequisites.sh`. If you haven’t got a local copy of the ESP-IDF yet, that’s OK - you can just download that one file in Raw format from here: [tools/windows/windows_install_prerequisites.sh](https://github.com/espressif/esp-idf/blob/master/tools/windows/windows_install_prerequisites.sh). Save it somewhere on your computer.
- Type the path to the shell script into the MSYS2 terminal window. You can type it as a normal Windows path, but use forward-slashes instead of back-slashes. ie: `C:/Users/myuser/Downloads/windows_install_prerequisites.sh`. You can read the script beforehand to check what it does.
- The `windows_install_prerequisites.sh` script will download and install packages for ESP-IDF support, and the ESP32 toolchain.

Troubleshooting

- While the install script runs, MSYS may update itself into a state where it can no longer operate. You may see errors like the following:

```
*** fatal error - cygheap base mismatch detected - 0x612E5408/0x612E4408. This_
↪problem is probably due to using incompatible versions of the cygwin DLL.
```

If you see errors like this, close the terminal window entirely (terminating the processes running there) and then re-open a new terminal. Re-run `windows_install_prerequisites.sh` (tip: use the up arrow key to see the last run command). The update process will resume after this step.

- MSYS2 is a “rolling” distribution so running the installer script may install newer packages than what is used in the prebuilt environments. If you see any errors that appear to be related to installing MSYS2 packages, please check the [MSYS2-packages issues list](#) for known issues. If you don't see any relevant issues, please [raise an IDF issue](#).

MSYS2 Mirrors in China There are some (unofficial) MSYS2 mirrors inside China, which substantially improves download speeds inside China.

To add these mirrors, edit the following two MSYS2 mirrorlist files before running the setup script. The mirrorlist files can be found in the `/etc/pacman.d` directory (i.e. `c:\msys2\etc\pacman.d`).

Add these lines at the top of `mirrorlist.mingw32`:

```
Server = https://mirrors.ustc.edu.cn/msys2/mingw/i686/
Server = http://mirror.bit.edu.cn/msys2/REPOS/MINGW/i686
```

Add these lines at the top of `mirrorlist.msys`:

```
Server = http://mirrors.ustc.edu.cn/msys2/msys/$arch
Server = http://mirror.bit.edu.cn/msys2/REPOS/MSYS2/$arch
```

HTTP Proxy You can enable an HTTP proxy for MSYS and PIP downloads by setting the `http_proxy` variable in the terminal before running the setup script:

```
export http_proxy='http://http.proxy.server:PORT'
```

Or with credentials:

```
export http_proxy='http://user:password@http.proxy.server:PORT'
```

Add this line to `/etc/profile` in the MSYS directory in order to permanently enable the proxy when using MSYS.

Alternative Setup: Just download a toolchain If you already have an MSYS2 install or want to do things differently, you can download just the toolchain here:

https://dl.espressif.com/dl/xtensa-esp32-elf-gcc8_4_0-esp-2020r3-win32.zip

Note: If you followed instructions *Configure Toolchain & Environment from Scratch*, you already have the toolchain and you won't need this download.

Important: Just having this toolchain is *not enough* to use ESP-IDF on Windows. You will need GNU make, bash, and sed at minimum. The above environments provide all this, plus a host compiler (required for menuconfig support).

Next Steps To carry on with development environment setup, proceed to section *Step 2. Get ESP-IDF*.

Updating The Environment When IDF is updated, sometimes new toolchains are required or new system requirements are added to the Windows MSYS2 environment.

Rather than setting up a new environment, you can update an existing Windows environment & toolchain:

- Update IDF to the new version you want to use.
- Run the `tools/windows/windows_install_prerequisites.sh` script inside IDF. This will install any new software packages that weren't previously installed, and download and replace the toolchain with the latest version.

The script to update MSYS2 may also fail with the same errors mentioned under *Troubleshooting*.

If you need to support multiple IDF versions concurrently, you can have different independent MSYS2 environments in different directories. Alternatively you can download multiple toolchains and unzip these to different directories, then use the PATH environment variable to set which one is the default.

Standard Setup of Toolchain for Linux (Legacy GNU Make)

Note: Since ESP-IDF V4.0, the default build system is based on CMake. This documentation is for the legacy build system based on GNU Make. Support for this build system may be removed in future major releases.

Install Prerequisites To compile with ESP-IDF you need to get the following packages:

- CentOS 7:

```
sudo yum install gcc git wget make flex bison gperf python python2-cryptography
```

- Ubuntu and Debian:

```
sudo apt-get install gcc git wget make flex bison gperf python python-pip  
↳python-setuptools python-serial python-cryptography python-future python-  
↳pyparsing python-pyelftools libffi-dev libssl-dev
```

- Arch:

```
sudo pacman -S --needed gcc git make flex bison gperf python-pyserial python-  
↳cryptography python-future python-pyparsing python-pyelftools
```

Note: Some older Linux distributions may be missing some of the Python packages listed above (or may use `pyserial` version 2.x which is not supported by ESP-IDF). It is possible to install these packages via `pip` instead - as described in section *Step 4. Install the Required Python Packages*.

Toolchain Setup ESP32 toolchain for Linux is available for download from Espressif website:

- for 64-bit Linux:
https://dl.espressif.com/dl/xtensa-esp32-elf-gcc8_4_0-esp-2020r3-linux-amd64.tar.gz
- for 32-bit Linux:
https://dl.espressif.com/dl/xtensa-esp32-elf-gcc8_4_0-esp-2020r3-linux-i686.tar.gz

1. Download this file, then extract it in `~/esp` directory:

- for 64-bit Linux:

```
mkdir -p ~/esp  
cd ~/esp  
tar -xzf ~/Downloads/xtensa-esp32-elf-gcc8_4_0-esp-2020r3-linux-amd64.tar.  
↳gz
```

- for 32-bit Linux:

```
mkdir -p ~/esp
cd ~/esp
tar -xzf ~/Downloads/xtensa-esp32-elf-gcc8_4_0-esp-2020r3-linux-i686.tar.gz
```

2. The toolchain will be extracted into `~/esp/xtensa-esp32-elf/` directory. To use it, you will need to update your `PATH` environment variable in `~/.profile` file. To make `xtensa-esp32-elf` available for all terminal sessions, add the following line to your `~/.profile` file:

```
export PATH="$HOME/esp/xtensa-esp32-elf/bin:$PATH"
```

Alternatively, you may create an alias for the above command. This way you can get the toolchain only when you need it. To do this, add different line to your `~/.profile` file:

```
alias get_esp32='export PATH="$HOME/esp/xtensa-esp32-elf/bin:$PATH"'
```

Then when you need the toolchain you can type `get_esp32` on the command line and the toolchain will be added to your `PATH`.

Note: If you have `/bin/bash` set as login shell, and both `.bash_profile` and `.profile` exist, then update `.bash_profile` instead. In CentOS, alias should set in `.bashrc`.

3. Log off and log in back to make the `.profile` changes effective. Run the following command to verify if `PATH` is correctly set:

```
printenv PATH
```

You are looking for similar result containing toolchain's path at the beginning of displayed string:

```
$ printenv PATH
/home/user-name/esp/xtensa-esp32-elf/bin:/home/user-name/bin:/home/user-name/.
↪local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/
↪games:/usr/local/games:/snap/bin
```

Instead of `/home/user-name` there should be a home path specific to your installation.

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

Next Steps To carry on with development environment setup, proceed to section [Step 2. Get ESP-IDF](#).

Related Documents

Setup Linux Toolchain from Scratch (Legacy GNU Make)

Note: Since ESP-IDF V4.0, the default build system is based on CMake. This documentation is for the legacy build system based on GNU Make. Support for this build system may be removed in future major releases.

Note: Standard process for installing the toolchain is described [here](#). See [Customized Setup of Toolchain](#) section for some of the reasons why installing the toolchain from scratch may be necessary.

Install Prerequisites To compile with ESP-IDF you need to get the following packages:

- Ubuntu and Debian:

```
sudo apt-get install gcc git wget make libncurses-dev flex bison gperf python_
↪python-pip python-setuptools python-serial python-cryptography python-future_
↪python-pyparsing python-pyelftools libffi-dev libssl-dev (continues on next page)
```

(continued from previous page)

- Arch:

```
sudo pacman -S --needed gcc git make ncurses flex bison gperf python-pyserial_
↳python-cryptography python-future python-pyparsing python-pyelftools
```

Note: Some older (pre-2014) Linux distributions may use `pyserial` version 2.x which is not supported by ESP-IDF. In this case please install a supported version via `pip` as it is described in section [Step 4. Install the Required Python Packages](#).

Compile the Toolchain from Source

- Install dependencies:

- CentOS 7:

```
sudo yum install gawk gperf grep gettext ncurses-devel python python-devel_
↳automake bison flex texinfo help2man libtool
```

- Ubuntu pre-16.04:

```
sudo apt-get install gawk gperf grep gettext libncurses-dev python python-
↳dev automake bison flex texinfo help2man libtool
```

- Ubuntu 16.04 or newer:

```
sudo apt-get install gawk gperf grep gettext python python-dev automake_
↳bison flex texinfo help2man libtool libtool-bin
```

- Debian 9:

```
sudo apt-get install gawk gperf grep gettext libncurses-dev python python-
↳dev automake bison flex texinfo help2man libtool libtool-bin
```

- Arch:

```
TODO
```

Create the working directory and go into it:

```
mkdir -p ~/esp
cd ~/esp
```

Download `crosstool-NG` and build it:

```
git clone https://github.com/espressif/crosstool-NG.git
cd crosstool-NG
git checkout esp-2020r3
git submodule update --init
./bootstrap && ./configure --enable-local && make
```

Build the toolchain:

```
./ct-ng xtensa-esp32-elf
./ct-ng build
chmod -R u+w builds/xtensa-esp32-elf
```

Toolchain will be built in `~/esp/crosstool-NG/builds/xtensa-esp32-elf`. Follow [instructions for standard setup](#) to add the toolchain to your `PATH`.

Next Steps To carry on with development environment setup, proceed to section [Step 2. Get ESP-IDF](#).

Standard Setup of Toolchain for Mac OS (Legacy GNU Make)

Note: Since ESP-IDF V4.0, the default build system is based on CMake. This documentation is for the legacy build system based on GNU Make. Support for this build system may be removed in future major releases.

Install Prerequisites

- install pip:

```
sudo easy_install pip
```

Note: pip will be used later for installing *the required Python packages*.

Toolchain Setup ESP32 toolchain for macOS is available for download from Espressif website:

https://dl.espressif.com/dl/xtensa-esp32-elf-gcc8_4_0-esp-2020r3-macos.tar.gz

Download this file, then extract it in ~/esp directory:

```
mkdir -p ~/esp
cd ~/esp
tar -xzf ~/Downloads/xtensa-esp32-elf-gcc8_4_0-esp-2020r3-macos.tar.gz
```

The toolchain will be extracted into ~/esp/xtensa-esp32-elf/ directory.

To use it, you will need to update your PATH environment variable in ~/.profile file. To make xtensa-esp32-elf available for all terminal sessions, add the following line to your ~/.profile file:

```
export PATH=$HOME/esp/xtensa-esp32-elf/bin:$PATH
```

Alternatively, you may create an alias for the above command. This way you can get the toolchain only when you need it. To do this, add different line to your ~/.profile file:

```
alias get_esp32="export PATH=$HOME/esp/xtensa-esp32-elf/bin:$PATH"
```

Then when you need the toolchain you can type get_esp32 on the command line and the toolchain will be added to your PATH.

Next Steps To carry on with development environment setup, proceed to section [Step 2. Get ESP-IDF](#).

Related Documents

Setup Toolchain for Mac OS from Scratch (Legacy GNU Make)

Note: Since ESP-IDF V4.0, the default build system is based on CMake. This documentation is for the legacy build system based on GNU Make. Support for this build system may be removed in future major releases.

Note: Standard process for installing the toolchain is described [here](#). See [Customized Setup of Toolchain](#) section for some of the reasons why installing the toolchain from scratch may be necessary.

Install Prerequisites

- install pip:

```
sudo easy_install pip
```

Note: pip will be used later for installing *the required Python packages*.

Compile the Toolchain from Source

- Install dependencies:
 - Install either [MacPorts](#) or [homebrew](#) package manager. MacPorts needs a full XCode installation, while homebrew only needs XCode command line tools.
 - with MacPorts:

```
sudo port install gsed gawk binutils gperf grep gettext wget libtool  
↪autoconf automake
```

- with homebrew:

```
brew install gnu-sed gawk binutils gperftools gettext wget help2man  
↪libtool autoconf automake
```

Create a case-sensitive filesystem image:

```
hdiutil create ~/esp/crosstool.dmg -volname "ctng" -size 10g -fs "Case-sensitive"  
↪HFS+
```

Mount it:

```
hdiutil mount ~/esp/crosstool.dmg
```

Create a symlink to your work directory:

```
mkdir -p ~/esp  
ln -s /Volumes/ctng ~/esp/ctng-volume
```

Go into the newly created directory:

```
cd ~/esp/ctng-volume
```

Download crosstool-NG and build it:

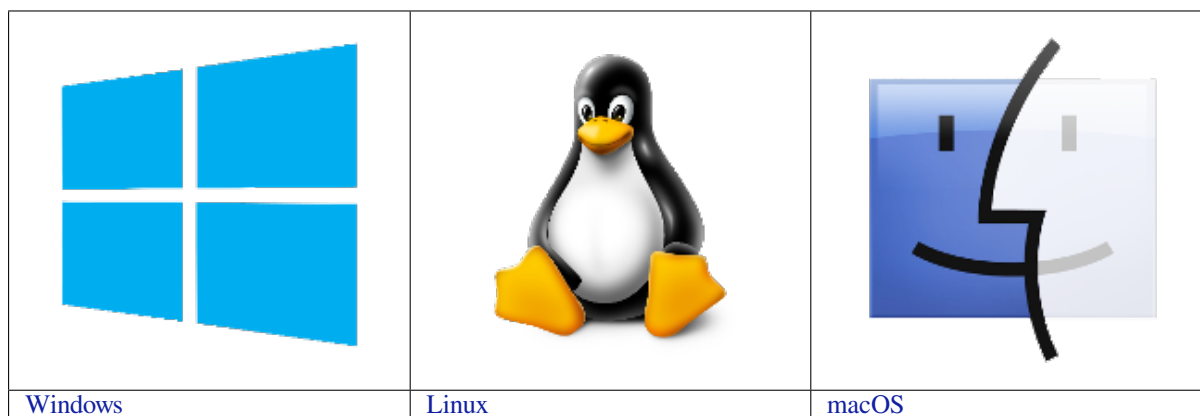
```
git clone https://github.com/espressif/crosstool-NG.git  
cd crosstool-NG  
git checkout esp-2020r3  
git submodule update --init  
./bootstrap && ./configure --enable-local && make
```

Build the toolchain:

```
./ct-ng xtensa-esp32-elf  
./ct-ng build  
chmod -R u+w builds/xtensa-esp32-elf
```

Toolchain will be built in `~/esp/ctng-volume/crosstool-NG/builds/xtensa-esp32-elf`. Follow [instructions for standard setup](#) to add the toolchain to your PATH.

Next Steps To carry on with development environment setup, proceed to section *Step 2. Get ESP-IDF*.



Note: This guide uses the directory `~/esp` on Linux and macOS or `%userprofile%\esp` on Windows as an installation folder for ESP-IDF. You can use any directory, but you will need to adjust paths for the commands respectively. Keep in mind that ESP-IDF does not support spaces in paths.

Depending on your experience and preferences, you may want to customize your environment instead of using a prebuilt toolchain. To set up the system your own way go to Section *Customized Setup of Toolchain (Legacy GNU Make)*.

Step 2. Get ESP-IDF

Besides the toolchain, you also need ESP32-specific API (software libraries and source code). They are provided by Espressif in [ESP-IDF repository](#).

To get a local copy of ESP-IDF, navigate to your installation directory and clone the repository with `git clone`.

Open Terminal, and run the following commands:

```
cd ~/esp
git clone -b v4.1.3 --recursive https://github.com/espressif/esp-idf.git
```

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

Consult [ESP-IDF Versions](#) for information about which ESP-IDF version to use in a given situation.

Note: The `git clone` option `-b v4.1.3` tells `git` to clone the tag in the ESP-IDF repository `git clone` corresponding to this version of the documentation.

Note: As a fallback, it is also possible to download a zip file of this stable release from the [Releases page](#). Do not download the “Source code” zip file(s) generated automatically by GitHub, they do not work with ESP-IDF.

Note: Do not miss the `--recursive` option. If you have already cloned ESP-IDF without this option, run another command to get all the submodules:

```
cd esp-idf
git submodule update --init
```

Step 3. Set Environment Variables

The toolchain uses the environment variable `IDF_PATH` to access the ESP-IDF directory. This variable should be set up on your computer, otherwise projects will not build.

These variables can be set temporarily (per session) or permanently. Please follow the instructions specific to [Windows](#), [Linux](#) and [macOS](#) in Section [Add IDF_PATH to User Profile \(Legacy GNU Make\)](#).

Step 4. Install the Required Python Packages

The python packages required by ESP-IDF are located in `IDF_PATH/requirements.txt`. You can install them by running:

```
python -m pip install --user -r $IDF_PATH/requirements.txt
```

Note: Please check the version of the Python interpreter that you will be using with ESP-IDF. For this, run the command `python --version` and depending on the result, you might want to use `python3`, `python3.7` or similar instead of just `python`, e.g.:

```
python3 -m pip install --user -r $IDF_PATH/requirements.txt
```

Step 5. Start a Project

Now you are ready to prepare your application for ESP32. You can start with [get-started/hello_world](#) project from [examples](#) directory in IDF.

Copy [get-started/hello_world](#) to the `~/esp` directory:

Linux and macOS

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

Windows

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

There is a range of example projects in the [examples](#) directory in ESP-IDF. You can copy any project in the same way as presented above and run it.

It is also possible to build examples in-place, without copying them first.

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

Step 6. Connect Your Device

Now connect your ESP32 board to the computer and check under what serial port the board is visible.

Serial ports have the following patterns in their names:

- **Windows:** names like `COM1`
- **Linux:** starting with `/dev/tty`
- **macOS:** starting with `/dev/cu`.

If you are not sure how to check the serial port name, please refer to [Establish Serial Connection with ESP32 \(Legacy GNU Make\)](#) for full details.

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

Step 7. Configure

Navigate to your `hello_world` directory from [Step 5. Start a Project](#) and run the project configuration utility `menuconfig`.

Linux and macOS

```
cd ~/esp/hello_world
make menuconfig
```

Windows

```
cd %userprofile%\esp\hello_world
make menuconfig
```

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

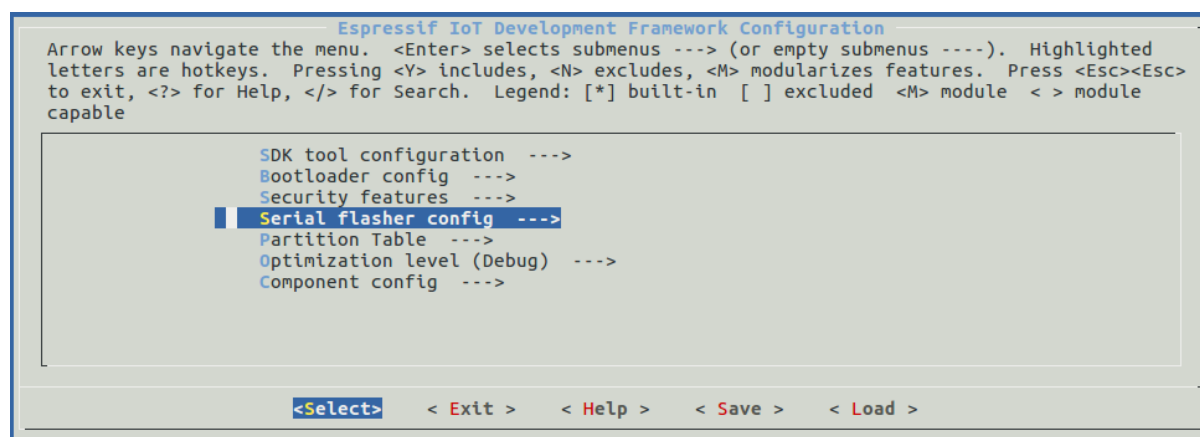


Fig. 41: Project configuration - Home window

In the menu, navigate to `Serial flasher config>Default serial port` to configure the serial port, where project will be loaded to. Confirm selection by pressing enter, save configuration by selecting `< Save >` and then exit `menuconfig` by selecting `< Exit >`.

To navigate and use `menuconfig`, press the following keys:

- Arrow keys for navigation
- Enter to go into a submenu
- Esc to go up one level or exit
- ? to see a help screen. Enter key exits the help screen
- Space, or Y and N keys to enable (Yes) and disable (No) configuration items with checkboxes “[*]”
- ? while highlighting a configuration item to display help about that item
- / to find configuration items

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

Step 8. Build and Flash

Build and flash the project by running:

```
make flash
```

This command will compile the application and all ESP-IDF components, then it will generate the bootloader, partition table, and application binaries. After that, these binaries will be flashed onto your ESP32 board.

If there are no issues by the end of the flash process, you will see messages (below) describing progress of the loading process. Then the board will be reset and the “hello_world” application will start up.

```
esptool.py v2.0-beta2
Flashing binaries to serial port /dev/ttyUSB0 (app at offset 0x10000)...
esptool.py v2.0-beta2
Connecting.....____
Uploading stub...
Running stub...
Stub running...
Changing baud rate to 921600
Changed.
Attaching SPI flash...
Configuring flash size...
Auto-detected Flash size: 4MB
Flash params set to 0x0220
Compressed 11616 bytes to 6695...
Wrote 11616 bytes (6695 compressed) at 0x00001000 in 0.1 seconds (effective 920.5
↳kbit/s)...
Hash of data verified.
Compressed 408096 bytes to 171625...
Wrote 408096 bytes (171625 compressed) at 0x00010000 in 3.9 seconds (effective 847.
↳3 kbit/s)...
Hash of data verified.
Compressed 3072 bytes to 82...
Wrote 3072 bytes (82 compressed) at 0x00008000 in 0.0 seconds (effective 8297.4
↳kbit/s)...
Hash of data verified.

Leaving...
Hard resetting...
```

If you’d like to use the Eclipse IDE instead of running `make`, check out the [Eclipse guide](#).

Step 9. Monitor

To check if “hello_world” is indeed running, type `make monitor`.

This command launches the *IDF Monitor* application:

```
$ make monitor
MONITOR
--- idf_monitor on /dev/ttyUSB0 115200 ---
--- Quit: Ctrl+] | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H ---
ets Jun  8 2016 00:22:57

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

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

```
...
Hello world!
Restarting in 10 seconds...
I (211) cpu_start: Starting scheduler on APP CPU.
Restarting in 9 seconds...
Restarting in 8 seconds...
Restarting in 7 seconds...
```

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

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

```
e000)(Xn@0y.!00(0PW+)00Hn9a~/90!0t500P0~0k00e0ea050jA
~zY00Y(10,1 00 e000)(Xn@0y.!Dr0zY(0 jpi0|0+z5Ymvp
```

If you have such a problem, do the following:

1. Exit the monitor.
2. Go back to [menuconfig](#).
3. Go to Component config → ESP32-specific → Main XTAL frequency, then change [CONFIG_ESP32_XTAL_FREQ_SEL](#) to 26MHz.
4. After that, [build and flash](#) the application again.

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

```
make flash monitor
```

See also [IDF Monitor](#) for handy shortcuts and more details on using IDF monitor.

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

Now you are ready to try some other [examples](#), or go straight to developing your own applications.

Environment Variables

Some environment variables can be specified whilst calling `make` allowing users to **override arguments without the need to reconfigure them using** `make menuconfig`.

| Variables | Description & Usage |
|---------------|--|
| ESP-PORT | Overrides the serial port used in <code>flash</code> and <code>monitor</code> . Examples: <code>make flash ESPPORT=/dev/ttyUSB1</code> , <code>make monitor ESPPORT=COM1</code> |
| ESP-BAUD | Overrides the serial baud rate when flashing the ESP32. Example: <code>make flash ESPBAUD=9600</code> |
| MON-ITOR-BAUD | Overrides the serial baud rate used when monitoring. Example: <code>make monitor MONITORBAUD=9600</code> |

Note: You can export environment variables (e.g. `export ESPPORT=/dev/ttyUSB1`). All subsequent calls of `make` within the same terminal session will use the exported value given that the variable is not simultaneously overridden.

Updating ESP-IDF

You should update ESP-IDF from time to time, as newer versions fix bugs and provide new features. The simplest way to do the update is to delete the existing `esp-idf` folder and clone it again, as if performing the initial installation described in [Step 2. Get ESP-IDF](#).

If downloading to a new path, remember to [Add IDF_PATH to User Profile \(Legacy GNU Make\)](#) so that the toolchain scripts can find ESP-IDF in its release specific location.

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

Related Documents

Add IDF_PATH to User Profile (Legacy GNU Make)

Note: Since ESP-IDF V4.0, the default build system is based on CMake. This documentation is for the legacy build system based on GNU Make. Support for this build system may be removed in future major releases.

To preserve setting of `IDF_PATH` environment variable between system restarts, add it to the user profile, following instructions below.

Windows The user profile scripts are contained in `C:/msys32/etc/profile.d/` directory. They are executed every time you open an MSYS2 window.

1. Create a new script file in `C:/msys32/etc/profile.d/` directory. Name it `export_idf_path.sh`.
2. Identify the path to ESP-IDF directory. It is specific to your system configuration and may look something like `C:\msys32\home\user-name\esp\esp-idf`
3. Add the `export` command to the script file, e.g.:

```
export IDF_PATH="C:/msys32/home/user-name/esp/esp-idf"
```

Remember to replace back-slashes with forward-slashes in the original Windows path.

4. Save the script file.
5. Close MSYS2 window and open it again. Check if `IDF_PATH` is set, by typing:

```
printenv IDF_PATH
```

The path previously entered in the script file should be printed out.

If you do not like to have `IDF_PATH` set up permanently in user profile, you should enter it manually on opening of an MSYS2 window:

```
export IDF_PATH="C:/msys32/home/user-name/esp/esp-idf"
```

If you got here from section [Step 3. Set Environment Variables](#), while installing s/w for ESP32 development, then go back to section [Step 5. Start a Project](#).

Linux and MacOS Set up `IDF_PATH` by adding the following line to `~/.profile` file:

```
export IDF_PATH=~/.esp/esp-idf
```

Log off and log in back to make this change effective.

Note: If you have `/bin/bash` set as login shell, and both `.bash_profile` and `.profile` exist, then update `.bash_profile` instead.

Run the following command to check if `IDF_PATH` is set:

```
printenv IDF_PATH
```

The path previously entered in `~/.profile` file (or set manually) should be printed out.

If you do not like to have `IDF_PATH` set up permanently, you should enter it manually in terminal window on each restart or logout:

```
export IDF_PATH=~/.esp/esp-idf
```

If you got here from section *Step 3. Set Environment Variables*, while installing s/w for ESP32 development, then go back to section *Step 5. Start a Project*.

Establish Serial Connection with ESP32 (Legacy GNU Make)

Note: Since ESP-IDF V4.0, the default build system is based on CMake. This documentation is for the legacy build system based on GNU Make. Support for this build system may be removed in future major releases.

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

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

Below are the links to drivers for ESP32 and other boards produced by Espressif:

| Development Board | USB Driver | Remarks |
|-------------------------------------|------------------------|-----------------------------------|
| ESP32-DevKitC | CP210x | |
| ESP32-LyraT | CP210x | |
| ESP32-LyraTD-MSC | CP210x | |
| ESP32-PICO-KIT | CP210x | |
| ESP-WROVER-KIT | FTDI | |
| ESP32 Demo Board | FTDI | |
| ESP-Prog | FTDI | Programmer board (w/o ESP32) |
| ESP32-MeshKit-Sense | n/a | Use with ESP-Prog |
| ESP32-Sense Kit | n/a | Use with ESP-Prog |

- [CP210x: CP210x USB to UART Bridge VCP Drivers](#)
- [FTDI: FTDI Virtual COM Port Drivers](#)

The drivers above are primarily for reference. Under normal circumstances, the drivers should be bundled with and operating system and automatically installed upon connecting one of the listed boards to the PC.

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

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

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

Linux

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

MacOS

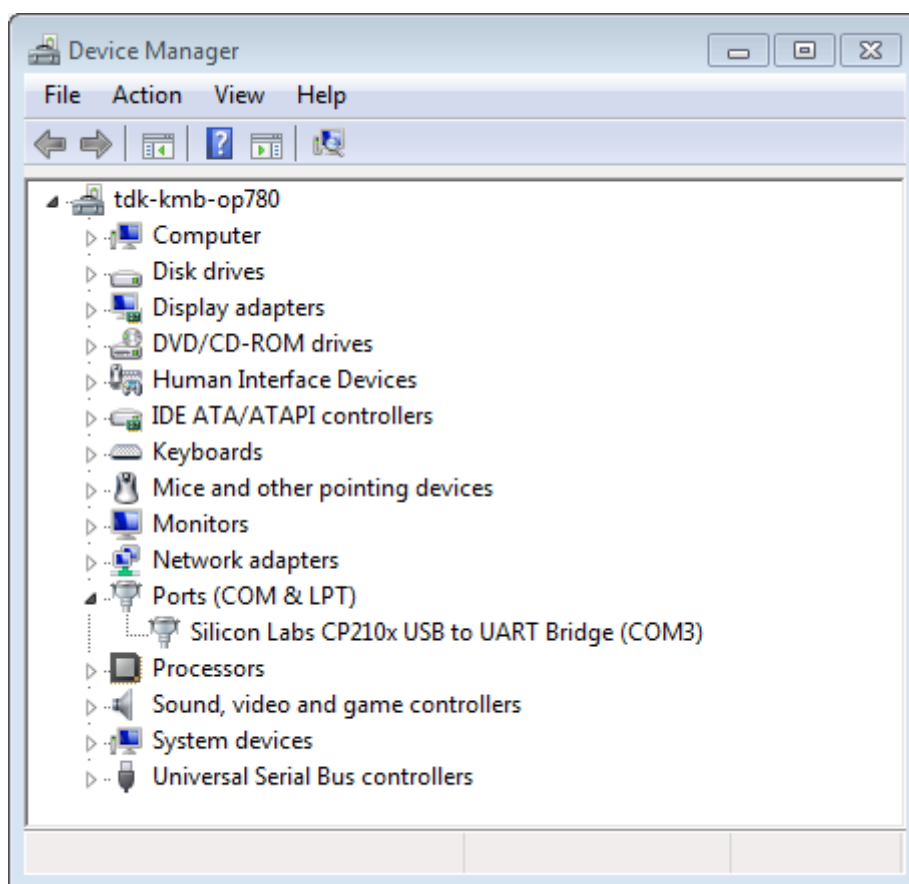


Fig. 42: USB to UART bridge of ESP32-DevKitC in Windows Device Manager

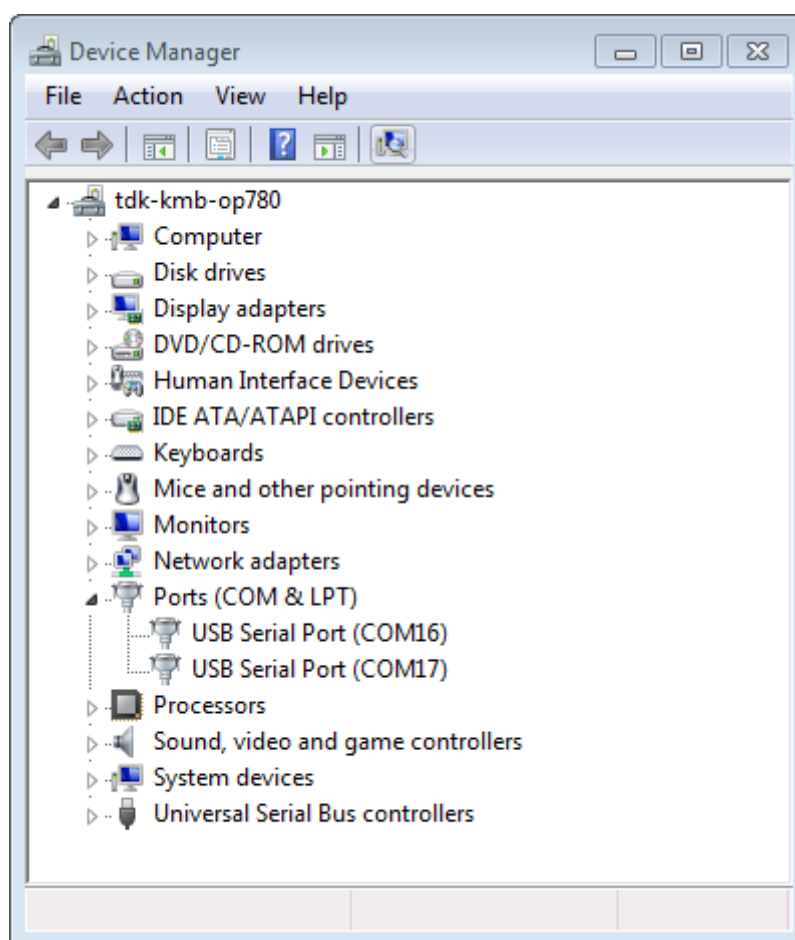


Fig. 43: Two USB Serial Ports of ESP-WROVER-KIT in Windows Device Manager

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

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

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

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

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

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

Verify serial connection Now verify that the serial connection is operational. You can do this using a serial terminal program. In this example we will use [PuTTY SSH Client](#) that is available for both Windows and Linux. You can use other serial program and set communication parameters like below.

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

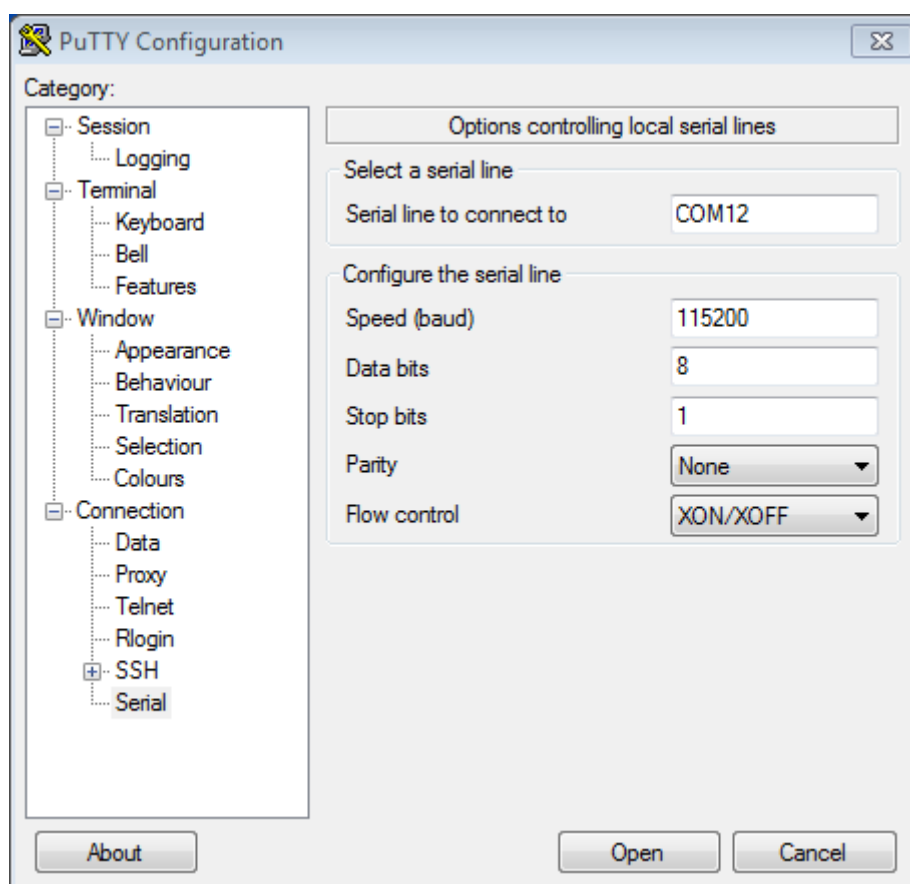


Fig. 44: Setting Serial Communication in PuTTY on Windows

Then open serial port in terminal and check, if you see any log printed out by ESP32. The log contents will depend on application loaded to ESP32. An example log by ESP32 is shown below.

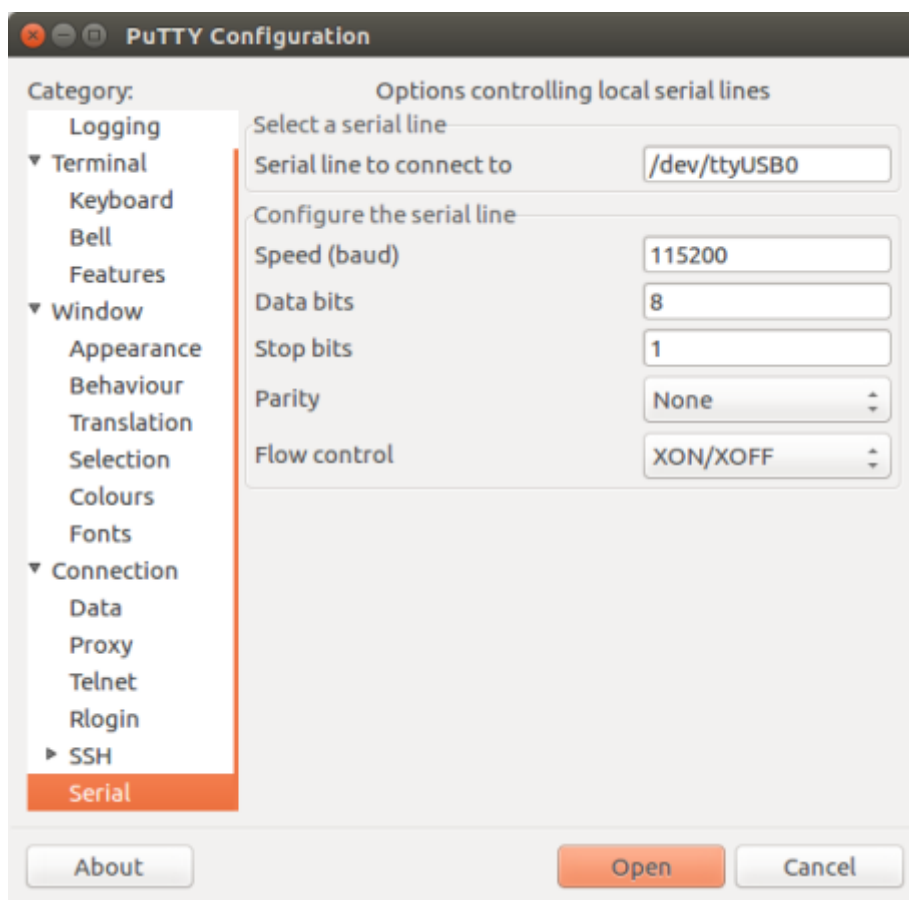


Fig. 45: Setting Serial Communication in PuTTY on Linux

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

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

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

...
```

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

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

Note: Close serial terminal after verification that communication is working. In next step we are going to use another application to upload ESP32. This application will not be able to access serial port while it is open in terminal.

If you got here from section [Step 6. Connect Your Device](#) when installing s/w for ESP32 development, then go back to section [Step 7. Configure](#).

Build and Flash with Make (Legacy GNU Make)

Note: Since ESP-IDF V4.0, the default build system is based on CMake. This documentation is for the legacy build system based on GNU Make. Support for this build system may be removed in future major releases.

Finding a project As well as the [esp-idf-template](#) project, ESP-IDF comes with some example projects on github in the [examples](#) directory.

Once you've found the project you want to work with, change to its directory and you can configure and build it.

Configuring your project

```
make menuconfig
```

Compiling your project

```
make all
```

...will compile app, bootloader and generate a partition table based on the config.

Flashing your project When `make all` finishes, it will print a command line to use `esptool.py` to flash the chip. However you can also do this from `make` by running:

```
make flash
```

This will flash the entire project (app, bootloader and partition table) to a new chip. Also if partition table has `ota_data` then this command will flash a initial `ota_data`. It allows to run the newly loaded app from a factory partition (or the first OTA partition, if factory partition is not present). The settings for serial port flashing can be configured with `make menuconfig`.

You don't need to run `make all` before running `make flash`, `make flash` will automatically rebuild anything which needs it.

Compiling & Flashing Just the App After the initial flash, you may just want to build and flash just your app, not the bootloader and partition table:

- `make app` - build just the app.
- `make app-flash` - flash just the app.

`make app-flash` will automatically rebuild the app if it needs it.

There's no downside to reflashing the bootloader and partition table each time, if they haven't changed.

The Partition Table Once you've compiled your project, the "build" directory will contain a binary file with a name like "my_app.bin". This is an ESP32 image binary that can be loaded by the bootloader.

A single ESP32's flash can contain multiple apps, as well as many kinds of data (calibration data, filesystems, parameter storage, etc). For this reason, a partition table is flashed to offset 0x8000 in the flash.

Each entry in the partition table has a name (label), type (app, data, or something else), subtype and the offset in flash where the partition is loaded.

The simplest way to use the partition table is to `make menuconfig` and choose one of the simple predefined partition tables:

- "Single factory app, no OTA"
- "Factory app, two OTA definitions"

In both cases the factory app is flashed at offset 0x10000. If you `make partition_table` then it will print a summary of the partition table.

For more details about *partition tables* and how to create custom variations, view the *documentation*.

Build and Flash with Eclipse IDE (Legacy GNU Make)

Note: Since ESP-IDF V4.0, the default build system is based on CMake. This documentation is for the legacy build system based on GNU Make. Support for this build system may be removed in future major releases.

Installing Eclipse IDE The Eclipse IDE gives you a graphical integrated development environment for writing, compiling and debugging ESP-IDF projects.

- Start by installing the esp-idf for your platform (see files in this directory with steps for Windows, OS X, Linux).
- We suggest building a project from the command line first, to get a feel for how that process works. You also need to use the command line to configure your esp-idf project (via `make menuconfig`), this is not currently supported inside Eclipse.
- Download the Eclipse Installer for your platform from eclipse.org.
- When running the Eclipse Installer, choose "Eclipse for C/C++ Development" (in other places you'll see this referred to as CDT.)

Setting up Eclipse Once your new Eclipse installation launches, follow these steps:

Import New Project

- Eclipse makes use of the Makefile support in ESP-IDF. This means you need to start by creating an ESP-IDF project. You can use the idf-template project from github, or open one of the examples in the esp-idf examples subdirectory.
- Once Eclipse is running, choose File -> Import...
- In the dialog that pops up, choose “C/C++” -> “Existing Code as Makefile Project” and click Next.
- On the next page, enter “Existing Code Location” to be the directory of your IDF project. Don't specify the path to the ESP-IDF directory itself (that comes later). The directory you specify should contain a file named “Makefile” (the project Makefile).
- On the same page, under “Toolchain for Indexer Settings” choose “Cross GCC” . Then click Finish.

Project Properties

- The new project will appear under Project Explorer. Right-click the project and choose Properties from the context menu.
- Click on the “Environment” properties page under “C/C++ Build” . Click “Add...” and enter name BATCH_BUILD and value 1.
- Click “Add...” again, and enter name IDF_PATH. The value should be the full path where ESP-IDF is installed. Windows users can copy the IDF_PATH from windows explorer.
- Edit the PATH environment variable. Keep the current value, and append the path to the Xtensa toolchain installed as part of IDF setup, if this is not already listed on the PATH. A typical path to the toolchain looks like /home/user-name/esp/xtensa-esp32-elf/bin. Note that you need to add a colon : before the appended path. Windows users will need to prepend C:\msys32\mingw32\bin;C:\msys32\opt\xtensa-esp32-elf\bin;C:\msys32\usr\bin to PATH environment variable (If you installed msys32 to a different directory then you'll need to change these paths to match).
- On macOS, add a PYTHONPATH environment variable and set it to /Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages. This is so that the system Python, which has pyserial installed as part of the setup steps, overrides any built-in Eclipse Python.

ADDITIONAL NOTE: If either the IDF_PATH directory or the project directory is located outside C:\msys32\home directory, you will have to give custom build command in C/C++ Build properties as: `python ${IDF_PATH}/tools/windows/eclipse_make.py` (Please note that the build time may get significantly increased by this method.)

Navigate to “C/C++ General” -> “Preprocessor Include Paths” property page:

- Click the “Providers” tab
- In the list of providers, click “CDT Cross GCC Built-in Compiler Settings” . Change “Command to get compiler specs” to `xtensa-esp32-elf-gcc ${FLAGS} -std=c++11 -E -P -v -dD "${INPUTS}”`.
- In the list of providers, click “CDT GCC Build Output Parser” and change the “Compiler command pattern” to `xtensa-esp32-elf-(gcc|g\+\+|c\+\+|cc|cpp|clang)`

Navigate to “C/C++ General” -> “Indexer” property page:

- Check “Enable project specific settings” to enable the rest of the settings on this page.
- Uncheck “Allow heuristic resolution of includes” . When this option is enabled Eclipse sometimes fails to find correct header directories.

Navigate to “C/C++ Build” -> “Behavior” property page:

- Check “Enable parallel build” to enable multiple build jobs in parallel.

Building in Eclipse Before your project is first built, Eclipse may show a lot of errors and warnings about undefined values. This is because some source files are automatically generated as part of the esp-idf build process. These errors and warnings will go away after you build the project.

- Click OK to close the Properties dialog in Eclipse.
- Outside Eclipse, open a command line prompt. Navigate to your project directory, and run `make menuconfig` to configure your project's esp-idf settings. This step currently has to be run outside Eclipse.

If you try to build without running a configuration step first, esp-idf will prompt for configuration on the command line - but Eclipse is not able to deal with this, so the build will hang or fail.

- Back in Eclipse, choose Project -> Build to build your project.

TIP: If your project had already been built outside Eclipse, you may need to do a Project -> Clean before choosing Project -> Build. This is so Eclipse can see the compiler arguments for all source files. It uses these to determine the header include paths.

Flash from Eclipse You can integrate the “make flash” target into your Eclipse project to flash using esptool.py from the Eclipse UI:

- Right-click your project in Project Explorer (important to make sure you select the project, not a directory in the project, or Eclipse may find the wrong Makefile.)
- Select Build Targets -> Create... from the context menu.
- Type “flash” as the target name. Leave the other options as their defaults.
- Now you can use Project -> Build Target -> Build (Shift+F9) to build the custom flash target, which will compile and flash the project.

Note that you will need to use “make menuconfig” to set the serial port and other config options for flashing. “make menuconfig” still requires a command line terminal (see the instructions for your platform.)

Follow the same steps to add `bootloader` and `partition_table` targets, if necessary.

Customized Setup of Toolchain (Legacy GNU Make) Instead of downloading binary toolchain from Espressif website (see [Step 1. Set up the Toolchain](#)) you may build the toolchain yourself.

Note: Since ESP-IDF V4.0, the default build system is based on CMake. This documentation is for the legacy build system based on GNU Make. Support for this build system may be removed in future major releases.

If you can't think of a reason why you need to build it yourself, then probably it's better to stick with the binary version. However, here are some of the reasons why you might want to compile it from source:

- if you want to customize toolchain build configuration
- if you want to use a different GCC version (such as 4.8.5)
- if you want to hack gcc or newlib or libstdc++
- if you are curious and/or have time to spare
- if you don't trust binaries downloaded from the Internet

In any case, here are the instructions to compile the toolchain yourself.

Chapter 2

API Reference

2.1 Bluetooth API

2.1.1 Controller && VHCI

Overview

Instructions

Application Example

Check `bluetooth/bluedroid/hci` folder in ESP-IDF examples, which contains the following application:

- This is a BLE advertising demo with virtual HCI interface. Send `Reset/ADV_PARAM/ADV_DATA/ADV_ENABLE` HCI command for BLE advertising - [bluetooth/bluedroid/hci/controller_vhci_ble_adv](#).

API Reference

Header File

- `bt/include/esp_bt.h`

Functions

`esp_err_t esp_ble_tx_power_set (esp_ble_power_type_t power_type, esp_power_level_t power_level)`

Set BLE TX power Connection Tx power should only be set after connection created.

Return ESP_OK - success, other - failed

Parameters

- `power_type`: : The type of which tx power, could set Advertising/Connection/Default and etc
- `power_level`: Power level(index) corresponding to absolute value(dbm)

`esp_power_level_t esp_ble_tx_power_get (esp_ble_power_type_t power_type)`

Get BLE TX power Connection Tx power should only be get after connection created.

Return ≥ 0 - Power level, < 0 - Invalid

Parameters

- `power_type`: : The type of which tx power, could set Advertising/Connection/Default and etc

`esp_err_t esp_bredr_tx_power_set (esp_power_level_t min_power_level, esp_power_level_t max_power_level)`

Set BR/EDR TX power BR/EDR power control will use the power in range of minimum value and maximum value. The power level will effect the global BR/EDR TX power, such inquire, page, connection and so on.

Please call the function after `esp_bt_controller_enable` and before any function which cause RF do TX. So you can call the function before doing discovery, profile init and so on. For example, if you want BR/EDR use the new TX power to do inquire, you should call this function before inquire. Another word, If call this function when BR/EDR is in inquire(ING), please do inquire again after call this function. Default minimum power level is `ESP_PWR_LVL_N0`, and maximum power level is `ESP_PWR_LVL_P3`.

Return `ESP_OK` - success, other - failed

Parameters

- `min_power_level`: The minimum power level
- `max_power_level`: The maximum power level

esp_err_t **esp_bredr_tx_power_get** (*esp_power_level_t* *`min_power_level`, *esp_power_level_t* *`max_power_level`)

Get BR/EDR TX power If the argument is not NULL, then store the corresponding value.

Return `ESP_OK` - success, other - failed

Parameters

- `min_power_level`: The minimum power level
- `max_power_level`: The maximum power level

esp_err_t **esp_bredr_sco_datapath_set** (*esp_sco_data_path_t* `data_path`)

set default SCO data path Should be called after controller is enabled, and before (e)SCO link is established

Return `ESP_OK` - success, other - failed

Parameters

- `data_path`: SCO data path

esp_err_t **esp_bt_controller_init** (*esp_bt_controller_config_t* *`cfg`)

Initialize BT controller to allocate task and other resource. This function should be called only once, before any other BT functions are called.

Return `ESP_OK` - success, other - failed

Parameters

- `cfg`: Initial configuration of BT controller. Different from previous version, there' s a mode and some connection configuration in “`cfg`” to configure controller work mode and allocate the resource which is needed.

esp_err_t **esp_bt_controller_deinit** (void)

De-initialize BT controller to free resource and delete task.

This function should be called only once, after any other BT functions are called.

Return `ESP_OK` - success, other - failed

esp_err_t **esp_bt_controller_enable** (*esp_bt_mode_t* `mode`)

Enable BT controller. Due to a known issue, you cannot call `esp_bt_controller_enable()` a second time to change the controller mode dynamically. To change controller mode, call `esp_bt_controller_disable()` and then call `esp_bt_controller_enable()` with the new mode.

Return `ESP_OK` - success, other - failed

Parameters

- `mode`: : the mode(BLE/BT/BTDM) to enable. For compatible of API, retain this argument. This mode must be equal as the mode in “`cfg`” of `esp_bt_controller_init()`.

esp_err_t **esp_bt_controller_disable** (void)

Disable BT controller.

Return `ESP_OK` - success, other - failed

esp_bt_controller_status_t **esp_bt_controller_get_status** (void)

Get BT controller is initialised/de-initialised/enabled/disabled.

Return status value

bool **esp_vhci_host_check_send_available** (void)

`esp_vhci_host_check_send_available` used for check actively if the host can send packet to controller or not.

Return true for ready to send, false means cannot send packet

void **esp_vhci_host_send_packet** (uint8_t *data, uint16_t len)
esp_vhci_host_send_packet host send packet to controller

Should not call this function from within a critical section or when the scheduler is suspended.

Parameters

- data: the packet point
- len: the packet length

esp_err_t **esp_vhci_host_register_callback** (const *esp_vhci_host_callback_t* *callback)
esp_vhci_host_register_callback register the vhci reference callback struct defined by vhci_host_callback structure.

Return ESP_OK - success, ESP_FAIL - failed

Parameters

- callback: *esp_vhci_host_callback* type variable

esp_err_t **esp_bt_controller_mem_release** (*esp_bt_mode_t* mode)

esp_bt_controller_mem_release release the controller memory as per the mode

This function releases the BSS, data and other sections of the controller to heap. The total size is about 70k bytes.

esp_bt_controller_mem_release(mode) should be called only before esp_bt_controller_init() or after esp_bt_controller_deinit().

Note that once BT controller memory is released, the process cannot be reversed. It means you cannot use the bluetooth mode which you have released by this function.

If your firmware will later upgrade the Bluetooth controller mode (BLE -> BT Classic or disabled -> enabled) then do not call this function.

If the app calls esp_bt_controller_enable(ESP_BT_MODE_BLE) to use BLE only then it is safe to call esp_bt_controller_mem_release(ESP_BT_MODE_CLASSIC_BT) at initialization time to free unused BT Classic memory.

If the mode is ESP_BT_MODE_BTDM, then it may be useful to call API esp_bt_mem_release(ESP_BT_MODE_BTDM) instead, which internally calls esp_bt_controller_mem_release(ESP_BT_MODE_BTDM) and additionally releases the BSS and data consumed by the BT/BLE host stack to heap. For more details about usage please refer to the documentation of esp_bt_mem_release() function

Return ESP_OK - success, other - failed

Parameters

- mode: : the mode want to release memory

esp_err_t **esp_bt_mem_release** (*esp_bt_mode_t* mode)

esp_bt_mem_release release controller memory and BSS and data section of the BT/BLE host stack as per the mode

This function first releases controller memory by internally calling esp_bt_controller_mem_release(). Additionally, if the mode is set to ESP_BT_MODE_BTDM, it also releases the BSS and data consumed by the BT/BLE host stack to heap

Note that once BT memory is released, the process cannot be reversed. It means you cannot use the bluetooth mode which you have released by this function.

If your firmware will later upgrade the Bluetooth controller mode (BLE -> BT Classic or disabled -> enabled) then do not call this function.

If you never intend to use bluetooth in a current boot-up cycle, you can call esp_bt_mem_release(ESP_BT_MODE_BTDM) before esp_bt_controller_init or after esp_bt_controller_deinit.

For example, if a user only uses bluetooth for setting the WiFi configuration, and does not use bluetooth in the rest of the product operation” . In such cases, after receiving the WiFi configuration, you can disable/deinit bluetooth and release its memory. Below is the sequence of APIs to be called for such scenarios:

```
esp_bluedroid_disable();
esp_bluedroid_deinit();
esp_bt_controller_disable();
esp_bt_controller_deinit();
esp_bt_mem_release(ESP_BT_MODE_BTDM);
```

Note In case of NimBLE host, to release BSS and data memory to heap, the mode needs to be set to ESP_BT_MODE_BTDM as controller is dual mode.

Return ESP_OK - success, other - failed

Parameters

- mode: : the mode whose memory is to be released

esp_err_t **esp_bt_sleep_enable**(void)

enable bluetooth to enter modem sleep

Note that this function shall not be invoked before esp_bt_controller_enable()

There are currently two options for bluetooth modem sleep, one is ORIG mode, and another is EVED Mode. EVED Mode is intended for BLE only.

For ORIG mode: Bluetooth modem sleep is enabled in controller start up by default if CONFIG_BTDM_MODEM_SLEEP is set and “ORIG mode” is selected. In ORIG modem sleep mode, bluetooth controller will switch off some components and pause to work every now and then, if there is no event to process; and wakeup according to the scheduled interval and resume the work. It can also wakeup earlier upon external request using function “esp_bt_controller_wakeup_request” .

Return

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_scan_duplicate_list_flush**(void)

Manually clear scan duplicate list.

Note that scan duplicate list will be automatically cleared when the maximum amount of device in the filter is reached the amount of device in the filter can be configured in menuconfig.

Note This function name is incorrectly spelled, it will be fixed in release 5.x version.

Return

- ESP_OK : success
- other : failed

Structures

struct esp_bt_controller_config_t

Controller config options, depend on config mask. Config mask indicate which functions enabled, this means some options or parameters of some functions enabled by config mask.

Public Members

uint16_t **controller_task_stack_size**

Bluetooth controller task stack size

uint8_t **controller_task_prio**

Bluetooth controller task priority

uint8_t **hci_uart_no**

If use UART1/2 as HCI IO interface, indicate UART number

uint32_t **hci_uart_baudrate**

If use UART1/2 as HCI IO interface, indicate UART baudrate

`uint8_t scan_duplicate_mode`
scan duplicate mode

`uint8_t scan_duplicate_type`
scan duplicate type

`uint16_t normal_adv_size`
Normal adv size for scan duplicate

`uint16_t mesh_adv_size`
Mesh adv size for scan duplicate

`uint16_t send_adv_reserved_size`
Controller minimum memory value

`uint32_t controller_debug_flag`
Controller debug log flag

`uint8_t mode`
Controller mode: BR/EDR, BLE or Dual Mode

`uint8_t ble_max_conn`
BLE maximum connection numbers

`uint8_t bt_max_acl_conn`
BR/EDR maximum ACL connection numbers

`uint8_t bt_sco_datapath`
SCO data path, i.e. HCI or PCM module

`bool auto_latency`
BLE auto latency, used to enhance classic BT performance

`bool bt_legacy_auth_vs_evt`
BR/EDR Legacy auth complete event required to protect from BIAS attack

`uint8_t bt_max_sync_conn`
BR/EDR maximum ACL connection numbers. Effective in menuconfig

`uint8_t ble_sca`
BLE low power crystal accuracy index

`uint8_t pcm_role`
PCM role (master & slave)

`uint8_t pcm_polar`
PCM polar trig (falling clk edge & rising clk edge)

`bool hli`
Using high level interrupt or not

`uint32_t magic`
Magic number

struct esp_vhci_host_callback
esp_vhci_host_callback used for vhci call host function to notify what host need to do

Public Members

`void (*notify_host_send_available)` (void)
callback used to notify that the host can send packet to controller

`int (*notify_host_recv)` (uint8_t *data, uint16_t len)
callback used to notify that the controller has a packet to send to the host

Macros

ESP_BT_CONTROLLER_CONFIG_MAGIC_VAL
BT_CONTROLLER_INIT_CONFIG_DEFAULT ()

Type Definitions

typedef struct esp_vhci_host_callback esp_vhci_host_callback_t
esp_vhci_host_callback used for vhci call host function to notify what host need to do

Enumerations

enum esp_bt_mode_t
Bluetooth mode for controller enable/disable.

Values:

ESP_BT_MODE_IDLE = 0x00
Bluetooth is not running

ESP_BT_MODE_BLE = 0x01
Run BLE mode

ESP_BT_MODE_CLASSIC_BT = 0x02
Run Classic BT mode

ESP_BT_MODE_BTDM = 0x03
Run dual mode

enum [anonymous]

BLE sleep clock accuracy(SCA), values for ble_sca field in *esp_bt_controller_config_t*, currently only ESP_BLE_SCA_500PPM and ESP_BLE_SCA_250PPM are supported.

Values:

ESP_BLE_SCA_500PPM = 0
BLE SCA at 500ppm

ESP_BLE_SCA_250PPM
BLE SCA at 250ppm

ESP_BLE_SCA_150PPM
BLE SCA at 150ppm

ESP_BLE_SCA_100PPM
BLE SCA at 100ppm

ESP_BLE_SCA_75PPM
BLE SCA at 75ppm

ESP_BLE_SCA_50PPM
BLE SCA at 50ppm

ESP_BLE_SCA_30PPM
BLE SCA at 30ppm

ESP_BLE_SCA_20PPM
BLE SCA at 20ppm

enum esp_bt_controller_status_t
Bluetooth controller enable/disable/initialised/de-initialised status.

Values:

ESP_BT_CONTROLLER_STATUS_IDLE = 0

ESP_BT_CONTROLLER_STATUS_INITED

ESP_BT_CONTROLLER_STATUS_ENABLED

ESP_BT_CONTROLLER_STATUS_NUM

enum esp_ble_power_type_t

BLE tx power type ESP_BLE_PWR_TYPE_CONN_HDL0-8: for each connection, and only be set after connection completed. when disconnect, the correspond TX power is not effected. ESP_BLE_PWR_TYPE_ADV : for advertising/scan response. ESP_BLE_PWR_TYPE_SCAN : for scan. ESP_BLE_PWR_TYPE_DEFAULT : if each connection' s TX power is not set, it will use this default value. if neither in scan mode nor in adv mode, it will use this default value. If none of power type is set, system will use ESP_PWR_LVL_P3 as default for ADV/SCAN/CONN0-9.

Values:

ESP_BLE_PWR_TYPE_CONN_HDL0 = 0
For connection handle 0

ESP_BLE_PWR_TYPE_CONN_HDL1 = 1
For connection handle 1

ESP_BLE_PWR_TYPE_CONN_HDL2 = 2
For connection handle 2

ESP_BLE_PWR_TYPE_CONN_HDL3 = 3
For connection handle 3

ESP_BLE_PWR_TYPE_CONN_HDL4 = 4
For connection handle 4

ESP_BLE_PWR_TYPE_CONN_HDL5 = 5
For connection handle 5

ESP_BLE_PWR_TYPE_CONN_HDL6 = 6
For connection handle 6

ESP_BLE_PWR_TYPE_CONN_HDL7 = 7
For connection handle 7

ESP_BLE_PWR_TYPE_CONN_HDL8 = 8
For connection handle 8

ESP_BLE_PWR_TYPE_ADV = 9
For advertising

ESP_BLE_PWR_TYPE_SCAN = 10
For scan

ESP_BLE_PWR_TYPE_DEFAULT = 11
For default, if not set other, it will use default value

ESP_BLE_PWR_TYPE_NUM = 12
TYPE numbers

enum esp_power_level_t

Bluetooth TX power level(index), it' s just a index corresponding to power(dbm).

Values:

ESP_PWR_LVL_N12 = 0
Corresponding to -12dbm

ESP_PWR_LVL_N9 = 1
Corresponding to -9dbm

ESP_PWR_LVL_N6 = 2
Corresponding to -6dbm

ESP_PWR_LVL_N3 = 3
Corresponding to -3dbm

ESP_PWR_LVL_N0 = 4
Corresponding to 0dbm

ESP_PWR_LVL_P3 = 5

Corresponding to +3dbm

ESP_PWR_LVL_P6 = 6

Corresponding to +6dbm

ESP_PWR_LVL_P9 = 7

Corresponding to +9dbm

ESP_PWR_LVL_N14 = [ESP_PWR_LVL_N12](#)

Backward compatibility! Setting to -14dbm will actually result to -12dbm

ESP_PWR_LVL_N11 = [ESP_PWR_LVL_N9](#)

Backward compatibility! Setting to -11dbm will actually result to -9dbm

ESP_PWR_LVL_N8 = [ESP_PWR_LVL_N6](#)

Backward compatibility! Setting to -8dbm will actually result to -6dbm

ESP_PWR_LVL_N5 = [ESP_PWR_LVL_N3](#)

Backward compatibility! Setting to -5dbm will actually result to -3dbm

ESP_PWR_LVL_N2 = [ESP_PWR_LVL_N0](#)

Backward compatibility! Setting to -2dbm will actually result to 0dbm

ESP_PWR_LVL_P1 = [ESP_PWR_LVL_P3](#)

Backward compatibility! Setting to +1dbm will actually result to +3dbm

ESP_PWR_LVL_P4 = [ESP_PWR_LVL_P6](#)

Backward compatibility! Setting to +4dbm will actually result to +6dbm

ESP_PWR_LVL_P7 = [ESP_PWR_LVL_P9](#)

Backward compatibility! Setting to +7dbm will actually result to +9dbm

enum esp_sco_data_path_t

Bluetooth audio data transport path.

Values:

ESP_SCO_DATA_PATH_HCI = 0

data over HCI transport

ESP_SCO_DATA_PATH_PCM = 1

data over PCM interface

2.1.2 BT COMMON

BT GENERIC DEFINES

Overview [Instructions](#)

Application Example [Instructions](#)

API Reference

Header File

- [bt/host/bluedroid/api/include/api/esp_bt_defs.h](#)

Structures

struct esp_bt_uuid_t

UUID type.

Public Members

`uint16_t len`

UUID length, 16bit, 32bit or 128bit

`union esp_bt_uuid_t:[anonymous] uuid`

UUID

Macros

`ESP_BLUEDROID_STATUS_CHECK` (status)

`ESP_BT_OCTET16_LEN`

`ESP_BT_OCTET8_LEN`

`ESP_DEFAULT_GATT_IF`

Default GATT interface id.

`ESP_BLE_CONN_INT_MIN`

relate to `BTM_BLE_CONN_INT_MIN` in `stack/btm_ble_api.h`

`ESP_BLE_CONN_INT_MAX`

relate to `BTM_BLE_CONN_INT_MAX` in `stack/btm_ble_api.h`

`ESP_BLE_CONN_LATENCY_MAX`

relate to `ESP_BLE_CONN_LATENCY_MAX` in `stack/btm_ble_api.h`

`ESP_BLE_CONN_SUP_TOUT_MIN`

relate to `BTM_BLE_CONN_SUP_TOUT_MIN` in `stack/btm_ble_api.h`

`ESP_BLE_CONN_SUP_TOUT_MAX`

relate to `ESP_BLE_CONN_SUP_TOUT_MAX` in `stack/btm_ble_api.h`

`ESP_BLE_CONN_PARAM_UNDEF`

`ESP_BLE_SCAN_PARAM_UNDEF`

`ESP_BLE_IS_VALID_PARAM` (x, min, max)

Check the param is valid or not.

`ESP_UUID_LEN_16`

`ESP_UUID_LEN_32`

`ESP_UUID_LEN_128`

`ESP_BD_ADDR_LEN`

Bluetooth address length.

`ESP_BLE_ENC_KEY_MASK`

Used to exchange the encryption key in the init key & response key.

`ESP_BLE_ID_KEY_MASK`

Used to exchange the IRK key in the init key & response key.

`ESP_BLE_CSR_KEY_MASK`

Used to exchange the CSRK key in the init key & response key.

`ESP_BLE_LINK_KEY_MASK`

Used to exchange the link key(this key just used in the BLE & BR/EDR coexist mode) in the init key & response key.

`ESP_APP_ID_MIN`

Minimum of the application id.

`ESP_APP_ID_MAX`

Maximum of the application id.

`ESP_BD_ADDR_STR`

ESP_BD_ADDR_HEX (addr)

Type Definitions

```
typedef uint8_t esp_bt_octet16_t[ESP_BT_OCTET16_LEN]
```

```
typedef uint8_t esp_bt_octet8_t[ESP_BT_OCTET8_LEN]
```

```
typedef uint8_t esp_link_key[ESP_BT_OCTET16_LEN]
```

```
typedef uint8_t esp_bd_addr_t[ESP_BD_ADDR_LEN]
```

Bluetooth device address.

```
typedef uint8_t esp_ble_key_mask_t
```

Enumerations

```
enum esp_bt_status_t
```

Status Return Value.

Values:

```
ESP_BT_STATUS_SUCCESS = 0
```

```
ESP_BT_STATUS_FAIL
```

```
ESP_BT_STATUS_NOT_READY
```

```
ESP_BT_STATUS_NOMEM
```

```
ESP_BT_STATUS_BUSY
```

```
ESP_BT_STATUS_DONE = 5
```

```
ESP_BT_STATUS_UNSUPPORTED
```

```
ESP_BT_STATUS_PARM_INVALID
```

```
ESP_BT_STATUS_UNHANDLED
```

```
ESP_BT_STATUS_AUTH_FAILURE
```

```
ESP_BT_STATUS_RMT_DEV_DOWN = 10
```

```
ESP_BT_STATUS_AUTH_REJECTED
```

```
ESP_BT_STATUS_INVALID_STATIC_RAND_ADDR
```

```
ESP_BT_STATUS_PENDING
```

```
ESP_BT_STATUS_UNACCEPT_CONN_INTERVAL
```

```
ESP_BT_STATUS_PARAM_OUT_OF_RANGE
```

```
ESP_BT_STATUS_TIMEOUT
```

```
ESP_BT_STATUS_PEER_LE_DATA_LEN_UNSUPPORTED
```

```
ESP_BT_STATUS_CONTROL_LE_DATA_LEN_UNSUPPORTED
```

```
ESP_BT_STATUS_ERR_ILLEGAL_PARAMETER_FMT
```

```
ESP_BT_STATUS_MEMORY_FULL = 20
```

```
ESP_BT_STATUS_EIR_TOO_LARGE
```

```
enum esp_bt_dev_type_t
```

Bluetooth device type.

Values:

```
ESP_BT_DEVICE_TYPE_BREDR = 0x01
```

```
ESP_BT_DEVICE_TYPE_BLE = 0x02
```

```
ESP_BT_DEVICE_TYPE_DUMO = 0x03
```

enum esp_ble_addr_type_t

BLE device address type.

*Values:***BLE_ADDR_TYPE_PUBLIC** = 0x00**BLE_ADDR_TYPE_RANDOM** = 0x01**BLE_ADDR_TYPE_RPA_PUBLIC** = 0x02**BLE_ADDR_TYPE_RPA_RANDOM** = 0x03**enum esp_ble_wl_addr_type_t**

white list address type

*Values:***BLE_WL_ADDR_TYPE_PUBLIC** = 0x00**BLE_WL_ADDR_TYPE_RANDOM** = 0x01**BT MAIN API****Overview** [Instructions](#)**Application Example** [Instructions](#)**API Reference****Header File**

- [bt/host/bluedroid/api/include/api/esp_bt_main.h](#)

Functions*esp_err_t* **esp_bluedroid_get_status** (void)

Get bluetooth stack status.

Return Bluetooth stack status*esp_err_t* **esp_bluedroid_enable** (void)

Enable bluetooth, must after esp_bluedroid_init()

Return

- ESP_OK : Succeed
- Other : Failed

esp_err_t **esp_bluedroid_disable** (void)

Disable bluetooth, must prior to esp_bluedroid_deinit()

Return

- ESP_OK : Succeed
- Other : Failed

esp_err_t **esp_bluedroid_init** (void)

Init and alloc the resource for bluetooth, must be prior to every bluetooth stuff.

Return

- ESP_OK : Succeed
- Other : Failed

esp_err_t **esp_bluedroid_deinit** (void)

Deinit and free the resource for bluetooth, must be after every bluetooth stuff.

Return

- ESP_OK : Succeed
- Other : Failed

Enumerations**enum esp_btdev_status_t**

Bluetooth stack status type, to indicate whether the bluetooth stack is ready.

Values:

ESP_BTDEV_STATUS_UNINITIALIZED = 0

Bluetooth not initialized

ESP_BTDEV_STATUS_INITIALIZED

Bluetooth initialized but not enabled

ESP_BTDEV_STATUS_ENABLED

Bluetooth initialized and enabled

BT DEVICE APIs

Overview Bluetooth device reference APIs.

[Instructions](#)

Application Example [Instructions](#)

API Reference**Header File**

- [bt/host/bluetooth/api/include/api/esp_bt_device.h](#)

Functions

const uint8_t ***esp_bt_dev_get_address** (void)

Get bluetooth device address. Must use after “esp_btdev_enable” .

Return bluetooth device address (six bytes), or NULL if bluetooth stack is not enabled

esp_err_t **esp_bt_dev_set_device_name** (**const** char *name)

Set bluetooth device name. This function should be called after esp_btdev_enable() completes successfully. A BR/EDR/LE device type shall have a single Bluetooth device name which shall be identical irrespective of the physical channel used to perform the name discovery procedure.

Return

- ESP_OK : Succeed
- ESP_ERR_INVALID_ARG : if name is NULL pointer or empty, or string length out of limit
- ESP_ERR_INVALID_STATE : if bluetooth stack is not yet enabled
- ESP_FAIL : others

Parameters

- [in] name: : device name to be set

2.1.3 BT LE**GAP API**

Overview [Instructions](#)

Application Example Check `bluetooth/bluedroid/ble` folder in ESP-IDF examples, which contains the following demos and their tutorials:

- This is a SMP security client demo and its tutorial. This demo initiates its security parameters and acts as a GATT client, which can send a security request to the peer device and then complete the encryption procedure.
 - [bluetooth/bluedroid/ble/gatt_security_client](#)
 - [GATT Security Client Example Walkthrough](#)
- This is a SMP security server demo and its tutorial. This demo initiates its security parameters and acts as a GATT server, which can send a pair request to the peer device and then complete the encryption procedure.
 - [bluetooth/bluedroid/ble/gatt_security_server](#)
 - [GATT Security Server Example Walkthrough](#)

API Reference

Header File

- `bt/host/bluedroid/api/include/api/esp_gap_ble_api.h`

Functions

`esp_err_t esp_ble_gap_register_callback(esp_gap_ble_cb_t callback)`

This function is called to occur gap event, such as scan result.

Return

- ESP_OK : success
- other : failed

Parameters

- [in] `callback`: callback function

`esp_err_t esp_ble_gap_config_adv_data(esp_ble_adv_data_t *adv_data)`

This function is called to override the BTA default ADV parameters.

Return

- ESP_OK : success
- other : failed

Parameters

- [in] `adv_data`: Pointer to User defined ADV data structure. This memory space can not be freed until callback of `config_adv_data` is received.

`esp_err_t esp_ble_gap_set_scan_params(esp_ble_scan_params_t *scan_params)`

This function is called to set scan parameters.

Return

- ESP_OK : success
- other : failed

Parameters

- [in] `scan_params`: Pointer to User defined `scan_params` data structure. This memory space can not be freed until callback of `set_scan_params`

`esp_err_t esp_ble_gap_start_scanning(uint32_t duration)`

This procedure keep the device scanning the peer device which advertising on the air.

Return

- ESP_OK : success
- other : failed

Parameters

- [in] `duration`: Keeping the scanning time, the unit is second.

`esp_err_t esp_ble_gap_stop_scanning(void)`

This function call to stop the device scanning the peer device which advertising on the air.

Return

- ESP_OK : success

– other : failed

esp_err_t **esp_ble_gap_start_advertising** (*esp_ble_adv_params_t* *adv_params)

This function is called to start advertising.

Return

- ESP_OK : success
- other : failed

Parameters

- [in] adv_params: pointer to User defined adv_params data structure.

esp_err_t **esp_ble_gap_stop_advertising** (void)

This function is called to stop advertising.

Return

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_update_conn_params** (*esp_ble_conn_update_params_t* *params)

Update connection parameters, can only be used when connection is up.

Return

- ESP_OK : success
- other : failed

Parameters

- [in] params: - connection update parameters

esp_err_t **esp_ble_gap_set_pkt_data_len** (*esp_bd_addr_t* remote_device, uint16_t tx_data_length)

This function is to set maximum LE data packet size.

Return

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_set_rand_addr** (*esp_bd_addr_t* rand_addr)

This function sets the static Random Address and Non-Resolvable Private Address for the application.

Return

- ESP_OK : success
- other : failed

Parameters

- [in] rand_addr: the random address which should be setting

esp_err_t **esp_ble_gap_clear_rand_addr** (void)

This function clears the random address for the application.

Return

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gap_config_local_privacy** (bool privacy_enable)

Enable/disable privacy on the local device.

Return

- ESP_OK : success
- other : failed

Parameters

- [in] privacy_enable: - enable/disable privacy on remote device.

esp_err_t **esp_ble_gap_config_local_icon** (uint16_t icon)

set local gap appearance icon

Return

- ESP_OK : success
- other : failed

Parameters

- [in] `icon`: - External appearance value, these values are defined by the Bluetooth SIG, please refer to <https://www.bluetooth.com/specifications/gatt/viewer?attributeXmlFile=org.bluetooth.characteristic.gap.appearance.xml>

`esp_err_t esp_ble_gap_update_whitelist` (bool `add_remove`, `esp_bd_addr_t` `remote_bda`, `esp_ble_wl_addr_type_t` `wl_addr_type`)

Add or remove device from white list.

Return

- `ESP_OK` : success
- other : failed

Parameters

- [in] `add_remove`: the value is true if added the ble device to the white list, and false remove to the white list.
- [in] `remote_bda`: the remote device address add/remove from the white list.
- [in] `wl_addr_type`: whitelist address type

`esp_err_t esp_ble_gap_clear_whitelist` (void)

Clear all white list.

Return

- `ESP_OK` : success
- other : failed

`esp_err_t esp_ble_gap_get_whitelist_size` (uint16_t *`length`)

Get the whitelist size in the controller.

Return

- `ESP_OK` : success
- other : failed

Parameters

- [out] `length`: the white list length.

`esp_err_t esp_ble_gap_set_prefer_conn_params` (`esp_bd_addr_t` `bd_addr`, uint16_t `min_conn_int`, uint16_t `max_conn_int`, uint16_t `slave_latency`, uint16_t `supervision_tout`)

This function is called to set the preferred connection parameters when default connection parameter is not desired before connecting. This API can only be used in the master role.

Return

- `ESP_OK` : success
- other : failed

Parameters

- [in] `bd_addr`: BD address of the peripheral
- [in] `min_conn_int`: minimum preferred connection interval
- [in] `max_conn_int`: maximum preferred connection interval
- [in] `slave_latency`: preferred slave latency
- [in] `supervision_tout`: preferred supervision timeout

`esp_err_t esp_ble_gap_set_device_name` (const char *`name`)

Set device name to the local device.

Return

- `ESP_OK` : success
- other : failed

Parameters

- [in] `name`: - device name.

`esp_err_t esp_ble_gap_get_local_used_addr` (`esp_bd_addr_t` `local_used_addr`, uint8_t *`addr_type`)

This function is called to get local used address and address type. `uint8_t *esp_bt_dev_get_address(void)` get the public address.

Return - ESP_OK : success

- other : failed

Parameters

- [in] local_used_addr: - current local used ble address (six bytes)
- [in] addr_type: - ble address type

uint8_t *esp_ble_resolve_adv_data (uint8_t *adv_data, uint8_t type, uint8_t *length)

This function is called to get ADV data for a specific type.

Return pointer of ADV data

Parameters

- [in] adv_data: - pointer of ADV data which to be resolved
- [in] type: - finding ADV data type
- [out] length: - return the length of ADV data not including type

esp_err_t esp_ble_gap_config_adv_data_raw (uint8_t *raw_data, uint32_t raw_data_len)

This function is called to set raw advertising data. User need to fill ADV data by self.

Return

- ESP_OK : success
- other : failed

Parameters

- [in] raw_data: : raw advertising data
- [in] raw_data_len: : raw advertising data length , less than 31 bytes

esp_err_t esp_ble_gap_config_scan_rsp_data_raw (uint8_t *raw_data, uint32_t raw_data_len)

This function is called to set raw scan response data. User need to fill scan response data by self.

Return

- ESP_OK : success
- other : failed

Parameters

- [in] raw_data: : raw scan response data
- [in] raw_data_len: : raw scan response data length , less than 31 bytes

esp_err_t esp_ble_gap_read_rssi (esp_bd_addr_t remote_addr)

This function is called to read the RSSI of remote device. The address of link policy results are returned in the gap callback function with ESP_GAP_BLE_READ_RSSI_COMPLETE_EVT event.

Return

- ESP_OK : success
- other : failed

Parameters

- [in] remote_addr: : The remote connection device address.

esp_err_t esp_ble_gap_add_duplicate_scan_exceptional_device (esp_ble_duplicate_exceptional_info_type_t type, esp_duplicate_info_t device_info)

This function is called to add a device info into the duplicate scan exceptional list.

Return

- ESP_OK : success
- other : failed

Parameters

- [in] type: device info type, it is defined in esp_ble_duplicate_exceptional_info_type_t when type is MESH_BEACON_TYPE, MESH_PROV_SRV_ADV or MESH_PROXY_SRV_ADV , device_info is invalid.
- [in] device_info: the device information.

esp_err_t **esp_ble_gap_remove_duplicate_scan_exceptional_device** (*esp_ble_duplicate_exceptional_info_type_t* type, *esp_duplicate_info_t* device_info)

This function is called to remove a device info from the duplicate scan exceptional list.

Return

- ESP_OK : success
- other : failed

Parameters

- [in] type: device info type, it is defined in *esp_ble_duplicate_exceptional_info_type_t* when type is MESH_BEACON_TYPE, MESH_PROV_SRV_ADV or MESH_PROXY_SRV_ADV , device_info is invalid.
- [in] device_info: the device information.

esp_err_t **esp_ble_gap_clean_duplicate_scan_exceptional_list** (*esp_duplicate_scan_exceptional_list_type_t* list_type)

This function is called to clean the duplicate scan exceptional list. This API will delete all device information in the duplicate scan exceptional list.

Return

- ESP_OK : success
- other : failed

Parameters

- [in] list_type: duplicate scan exceptional list type, the value can be one or more of *esp_duplicate_scan_exceptional_list_type_t*.

esp_err_t **esp_ble_gap_set_security_param** (*esp_ble_sm_param_t* param_type, void *value, uint8_t len)

Set a GAP security parameter value. Overrides the default value.

Secure connection is highly recommended to avoid some major vulnerabilities like ‘Impersonation in the Pin Pairing Protocol’ (CVE-2020-26555) and ‘Authentication of the LE Legacy Pairing Protocol’ .

To accept only secure connection mode, it is necessary do as following:

1. Set bit ESP_LE_AUTH_REQ_SC_ONLY (param_type is ESP_BLE_SM_AUTHEN_REQ_MODE), bit ESP_LE_AUTH_BOND and bit ESP_LE_AUTH_REQ_MITM is optional as required.
2. Set to ESP_BLE_ONLY_ACCEPT_SPECIFIED_AUTH_ENABLE (param_type is ESP_BLE_SM_ONLY_ACCEPT_SPECIFIED_SEC_AUTH).

Return - ESP_OK : success

- other : failed

Parameters

- [in] param_type: : the type of the param which to be set
- [in] value: : the param value
- [in] len: : the length of the param value

esp_err_t **esp_ble_gap_security_rsp** (*esp_bd_addr_t* bd_addr, bool accept)

Grant security request access.

Return - ESP_OK : success

- other : failed

Parameters

- [in] bd_addr: : BD address of the peer
- [in] accept: : accept the security request or not

esp_err_t **esp_ble_set_encryption** (*esp_bd_addr_t* bd_addr, *esp_ble_sec_act_t* sec_act)

Set a gap parameter value. Use this function to change the default GAP parameter values.

Return - ESP_OK : success

- other : failed

Parameters

- [in] bd_addr: : the address of the peer device need to encryption

- [in] `sec_act`: This is the security action to indicate what kind of BLE security level is required for the BLE link if the BLE is supported

esp_err_t **esp_ble_passkey_reply** (*esp_bd_addr_t* `bd_addr`, bool `accept`, uint32_t `passkey`)

Reply the key value to the peer device in the legacy connection stage.

Return - ESP_OK : success

- other : failed

Parameters

- [in] `bd_addr`: : BD address of the peer
- [in] `accept`: : passkey entry successful or declined.
- [in] `passkey`: : passkey value, must be a 6 digit number, can be lead by 0.

esp_err_t **esp_ble_confirm_reply** (*esp_bd_addr_t* `bd_addr`, bool `accept`)

Reply the confirm value to the peer device in the secure connection stage.

Return - ESP_OK : success

- other : failed

Parameters

- [in] `bd_addr`: : BD address of the peer device
- [in] `accept`: : numbers to compare are the same or different.

esp_err_t **esp_ble_remove_bond_device** (*esp_bd_addr_t* `bd_addr`)

Removes a device from the security database list of peer device. It manages unpairing event while connected.

Return - ESP_OK : success

- other : failed

Parameters

- [in] `bd_addr`: : BD address of the peer device

int **esp_ble_get_bond_device_num** (void)

Get the device number from the security database list of peer device. It will return the device bonded number immediately.

Return - ≥ 0 : bonded devices number.

- ESP_FAIL : failed

esp_err_t **esp_ble_get_bond_device_list** (int *`dev_num`, *esp_ble_bond_dev_t* *`dev_list`)

Get the device from the security database list of peer device. It will return the device bonded information immediately.

Return - ESP_OK : success

- other : failed

Parameters

- [inout] `dev_num`: Indicate the `dev_list` array(buffer) size as input. If `dev_num` is large enough, it means the actual number as output. Suggest that `dev_num` value equal to `esp_ble_get_bond_device_num()`.
- [out] `dev_list`: an array(buffer) of *esp_ble_bond_dev_t* type. Use for storing the bonded devices address. The `dev_list` should be allocated by who call this API.

esp_err_t **esp_ble_oob_req_reply** (*esp_bd_addr_t* `bd_addr`, uint8_t *`TK`, uint8_t `len`)

This function is called to provide the OOB data for SMP in response to ESP_GAP_BLE_OOB_REQ_EVT.

Return - ESP_OK : success

- other : failed

Parameters

- [in] `bd_addr`: BD address of the peer device.
- [in] `TK`: TK value, the TK value shall be a 128-bit random number
- [in] `len`: length of tk, should always be 128-bit

esp_err_t **esp_ble_gap_disconnect** (*esp_bd_addr_t* `remote_device`)

This function is to disconnect the physical connection of the peer device gattc may have multiple virtual GATT server connections when multiple `app_id` registered. `esp_ble_gattc_close` (*esp_gatt_if_t* `gattc_if`, uint16_t

conn_id) only close one virtual GATT server connection. if there exist other virtual GATT server connections, it does not disconnect the physical connection. esp_ble_gap_disconnect(esp_bd_addr_t remote_device) disconnect the physical connection directly.

Return - ESP_OK : success

- other : failed

Parameters

- [in] remote_device: : BD address of the peer device

esp_err_t **esp_ble_get_current_conn_params** (*esp_bd_addr_t* bd_addr, *esp_gap_conn_params_t* *conn_params)

This function is called to read the connection parameters information of the device.

Return - ESP_OK : success

- other : failed

Parameters

- [in] bd_addr: BD address of the peer device.
- [out] conn_params: the connection parameters information

esp_err_t **esp_gap_ble_set_channels** (*esp_gap_ble_channels_t* channels)

BLE set channels.

Return - ESP_OK : success

- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- other : failed

Parameters

- [in] channels: : The n th such field (in the range 0 to 36) contains the value for the link layer channel index n. 0 means channel n is bad. 1 means channel n is unknown. The most significant bits are reserved and shall be set to 0. At least one channel shall be marked as unknown.

esp_err_t **esp_gap_ble_set_authorization** (*esp_bd_addr_t* bd_addr, bool authorize)

This function is called to authorized a link after Authentication(MITM protection)

Return - ESP_OK : success

- other : failed

Parameters

- [in] bd_addr: BD address of the peer device.
- [out] authorize: Authorized the link or not.

Unions

union esp_ble_key_value_t

#include <esp_gap_ble_api.h> union type of the security key value

Public Members

esp_ble_penc_keys_t **penc_key**

received peer encryption key

esp_ble_pcsrkeys_t **pcsrk_key**

received peer device SRK

esp_ble_pidkeys_t **pid_key**

peer device ID key

esp_ble_lenckeys_t **lenc_key**

local encryption reproduction keys LTK = d1(ER,DIV,0)

esp_ble_lcsrkeys_t **lcsrkey**

local device CSRK = d1(ER,DIV,1)

union esp_ble_sec_t

#include <esp_gap_ble_api.h> union associated with ble security

Public Members

esp_ble_sec_key_notif_t **key_notif**
passkey notification

esp_ble_sec_req_t **ble_req**
BLE SMP related request

esp_ble_key_t **ble_key**
BLE SMP keys used when pairing

esp_ble_local_id_keys_t **ble_id_keys**
BLE IR event

esp_ble_auth_cmpl_t **auth_cmpl**
Authentication complete indication.

union esp_ble_gap_cb_param_t
#include <esp_gap_ble_api.h> Gap callback parameters union.

Public Members

struct *esp_ble_gap_cb_param_t::ble_adv_data_cmpl_evt_param* **adv_data_cmpl**
Event parameter of ESP_GAP_BLE_ADV_DATA_SET_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t::ble_scan_rsp_data_cmpl_evt_param* **scan_rsp_data_cmpl**
Event parameter of ESP_GAP_BLE_SCAN_RSP_DATA_SET_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t::ble_scan_param_cmpl_evt_param* **scan_param_cmpl**
Event parameter of ESP_GAP_BLE_SCAN_PARAM_SET_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t::ble_scan_result_evt_param* **scan_rst**
Event parameter of ESP_GAP_BLE_SCAN_RESULT_EVT

struct *esp_ble_gap_cb_param_t::ble_adv_data_raw_cmpl_evt_param* **adv_data_raw_cmpl**
Event parameter of ESP_GAP_BLE_ADV_DATA_RAW_SET_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t::ble_scan_rsp_data_raw_cmpl_evt_param* **scan_rsp_data_raw_cmpl**
Event parameter of ESP_GAP_BLE_SCAN_RSP_DATA_RAW_SET_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t::ble_adv_start_cmpl_evt_param* **adv_start_cmpl**
Event parameter of ESP_GAP_BLE_ADV_START_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t::ble_scan_start_cmpl_evt_param* **scan_start_cmpl**
Event parameter of ESP_GAP_BLE_SCAN_START_COMPLETE_EVT

esp_ble_sec_t **ble_security**
ble gap security union type

struct *esp_ble_gap_cb_param_t::ble_scan_stop_cmpl_evt_param* **scan_stop_cmpl**
Event parameter of ESP_GAP_BLE_SCAN_STOP_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t::ble_adv_stop_cmpl_evt_param* **adv_stop_cmpl**
Event parameter of ESP_GAP_BLE_ADV_STOP_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t::ble_set_rand_cmpl_evt_param* **set_rand_addr_cmpl**
Event parameter of ESP_GAP_BLE_SET_STATIC_RAND_ADDR_EVT

struct *esp_ble_gap_cb_param_t::ble_update_conn_params_evt_param* **update_conn_params**
Event parameter of ESP_GAP_BLE_UPDATE_CONN_PARAMS_EVT

struct *esp_ble_gap_cb_param_t::ble_pkt_data_length_cmpl_evt_param* **pkt_data_lenth_cmpl**
Event parameter of ESP_GAP_BLE_SET_PKT_LENGTH_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t::ble_local_privacy_cmpl_evt_param* **local_privacy_cmpl**
Event parameter of ESP_GAP_BLE_SET_LOCAL_PRIVACY_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t::ble_remove_bond_dev_cmpl_evt_param* **remove_bond_dev_cmpl**
Event parameter of ESP_GAP_BLE_REMOVE_BOND_DEV_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t::ble_clear_bond_dev_cmpl_evt_param* **clear_bond_dev_cmpl**
Event parameter of ESP_GAP_BLE_CLEAR_BOND_DEV_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t::ble_get_bond_dev_cmpl_evt_param* **get_bond_dev_cmpl**
Event parameter of ESP_GAP_BLE_GET_BOND_DEV_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t::ble_read_rssi_cmpl_evt_param* **read_rssi_cmpl**
Event parameter of ESP_GAP_BLE_READ_RSSI_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t::ble_update_whitelist_cmpl_evt_param* **update_whitelist_cmpl**
Event parameter of ESP_GAP_BLE_UPDATE_WHITELIST_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t::ble_update_duplicate_exceptional_list_cmpl_evt_param* **update_duplicate_except**
Event parameter of ESP_GAP_BLE_UPDATE_DUPLICATE_EXCEPTIONAL_LIST_COMPLETE_EVT

struct *esp_ble_gap_cb_param_t::ble_set_channels_evt_param* **ble_set_channels**
Event parameter of ESP_GAP_BLE_SET_CHANNELS_EVT

struct **ble_adv_data_cmpl_evt_param**
#include <esp_gap_ble_api.h> ESP_GAP_BLE_ADV_DATA_SET_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**
Indicate the set advertising data operation success status

struct **ble_adv_data_raw_cmpl_evt_param**
#include <esp_gap_ble_api.h> ESP_GAP_BLE_ADV_DATA_RAW_SET_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**
Indicate the set raw advertising data operation success status

struct **ble_adv_start_cmpl_evt_param**
#include <esp_gap_ble_api.h> ESP_GAP_BLE_ADV_START_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**
Indicate advertising start operation success status

struct **ble_adv_stop_cmpl_evt_param**
#include <esp_gap_ble_api.h> ESP_GAP_BLE_ADV_STOP_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**
Indicate adv stop operation success status

struct **ble_clear_bond_dev_cmpl_evt_param**
#include <esp_gap_ble_api.h> ESP_GAP_BLE_CLEAR_BOND_DEV_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate the clear bond device operation success status

```
struct ble_get_bond_dev_cmpl_evt_param
#include <esp_gap_ble_api.h> ESP_GAP_BLE_GET_BOND_DEV_COMPLETE_EVT.
```

Public Members

esp_bt_status_t **status**

Indicate the get bond device operation success status

uint8_t **dev_num**

Indicate the get number device in the bond list

esp_ble_bond_dev_t ***bond_dev**

the pointer to the bond device Structure

```
struct ble_local_privacy_cmpl_evt_param
#include <esp_gap_ble_api.h> ESP_GAP_BLE_SET_LOCAL_PRIVACY_COMPLETE_EVT.
```

Public Members

esp_bt_status_t **status**

Indicate the set local privacy operation success status

```
struct ble_pkt_data_length_cmpl_evt_param
#include <esp_gap_ble_api.h> ESP_GAP_BLE_SET_PKT_LENGTH_COMPLETE_EVT.
```

Public Members

esp_bt_status_t **status**

Indicate the set pkt data length operation success status

esp_ble_pkt_data_length_params_t **params**

pkt data length value

```
struct ble_read_rssi_cmpl_evt_param
#include <esp_gap_ble_api.h> ESP_GAP_BLE_READ_RSSI_COMPLETE_EVT.
```

Public Members

esp_bt_status_t **status**

Indicate the read adv tx power operation success status

int8_t **rssi**

The ble remote device rssi value, the range is from -127 to 20, the unit is dbm, if the RSSI cannot be read, the RSSI metric shall be set to 127.

esp_bd_addr_t **remote_addr**

The remote device address

```
struct ble_remove_bond_dev_cmpl_evt_param
#include <esp_gap_ble_api.h> ESP_GAP_BLE_REMOVE_BOND_DEV_COMPLETE_EVT.
```

Public Members

esp_bt_status_t **status**

Indicate the remove bond device operation success status

esp_bd_addr_t **bd_addr**

The device address which has been remove from the bond list

struct ble_scan_param_cmpl_evt_param

#include <esp_gap_ble_api.h> ESP_GAP_BLE_SCAN_PARAM_SET_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate the set scan param operation success status

struct ble_scan_result_evt_param

#include <esp_gap_ble_api.h> ESP_GAP_BLE_SCAN_RESULT_EVT.

Public Members

esp_gap_search_evt_t **search_evt**

Search event type

esp_bd_addr_t **bda**

Bluetooth device address which has been searched

esp_bt_dev_type_t **dev_type**

Device type

esp_ble_addr_type_t **ble_addr_type**

Ble device address type

esp_ble_evt_type_t **ble_evt_type**

Ble scan result event type

int **rssi**

Searched device' s RSSI

uint8_t **ble_adv**[ESP_BLE_ADV_DATA_LEN_MAX + ESP_BLE_SCAN_RSP_DATA_LEN_MAX]

Received EIR

int **flag**

Advertising data flag bit

int **num_resps**

Scan result number

uint8_t **adv_data_len**

Adv data length

uint8_t **scan_rsp_len**

Scan response length

uint32_t **num_dis**

The number of discard packets

struct ble_scan_rsp_data_cmpl_evt_param

#include <esp_gap_ble_api.h> ESP_GAP_BLE_SCAN_RSP_DATA_SET_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**

Indicate the set scan response data operation success status

```
struct ble_scan_rsp_data_raw_cmpl_evt_param
#include <esp_gap_ble_api.h> ESP_GAP_BLE_SCAN_RSP_DATA_RAW_SET_COMPLETE_EVT.
```

Public Members

esp_bt_status_t **status**

Indicate the set raw advertising data operation success status

```
struct ble_scan_start_cmpl_evt_param
#include <esp_gap_ble_api.h> ESP_GAP_BLE_SCAN_START_COMPLETE_EVT.
```

Public Members

esp_bt_status_t **status**

Indicate scan start operation success status

```
struct ble_scan_stop_cmpl_evt_param
#include <esp_gap_ble_api.h> ESP_GAP_BLE_SCAN_STOP_COMPLETE_EVT.
```

Public Members

esp_bt_status_t **status**

Indicate scan stop operation success status

```
struct ble_set_channels_evt_param
#include <esp_gap_ble_api.h> ESP_GAP_BLE_SET_CHANNELS_EVT.
```

Public Members

esp_bt_status_t **stat**

BLE set channel status

```
struct ble_set_rand_cmpl_evt_param
#include <esp_gap_ble_api.h> ESP_GAP_BLE_SET_STATIC_RAND_ADDR_EVT.
```

Public Members

esp_bt_status_t **status**

Indicate set static rand address operation success status

```
struct ble_update_conn_params_evt_param
#include <esp_gap_ble_api.h> ESP_GAP_BLE_UPDATE_CONN_PARAMS_EVT.
```

Public Members

esp_bt_status_t **status**

Indicate update connection parameters success status

esp_bd_addr_t **bda**

Bluetooth device address

uint16_t min_int
Min connection interval

uint16_t max_int
Max connection interval

uint16_t latency
Slave latency for the connection in number of connection events. Range: 0x0000 to 0x01F3

uint16_t conn_int
Current connection interval

uint16_t timeout
Supervision timeout for the LE Link. Range: 0x000A to 0x0C80. Mandatory Range: 0x000A to 0x0C80 Time = N * 10 msec

struct ble_update_duplicate_exceptional_list_cmpl_evt_param
#include <esp_gap_ble_api.h> ESP_GAP_BLE_UPDATE_DUPLICATE_EXCEPTIONAL_LIST_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**
Indicate update duplicate scan exceptional list operation success status

uint8_t subcode
Define in *esp_bt_duplicate_exceptional_subcode_type_t*

uint16_t length
The length of *device_info*

esp_duplicate_info_t **device_info**
device information, when subcode is ESP_BLE_DUPLICATE_EXCEPTIONAL_LIST_CLEAN, the value is invalid

struct ble_update_whitelist_cmpl_evt_param
#include <esp_gap_ble_api.h> ESP_GAP_BLE_UPDATE_WHITELIST_COMPLETE_EVT.

Public Members

esp_bt_status_t **status**
Indicate the add or remove whitelist operation success status

esp_ble_wl_operation_t **wl_operation**
The value is ESP_BLE_WHITELIST_ADD if add address to whitelist operation success, ESP_BLE_WHITELIST_REMOVE if remove address from the whitelist operation success

Structures

struct esp_ble_adv_params_t
Advertising parameters.

Public Members

uint16_t adv_int_min
Minimum advertising interval for undirected and low duty cycle directed advertising. Range: 0x0020 to 0x4000 Default: N = 0x0800 (1.28 second) Time = N * 0.625 msec Time Range: 20 ms to 10.24 sec

uint16_t adv_int_max
Maximum advertising interval for undirected and low duty cycle directed advertising. Range: 0x0020 to 0x4000 Default: N = 0x0800 (1.28 second) Time = N * 0.625 msec Time Range: 20 ms to 10.24 sec Advertising max interval

esp_ble_adv_type_t **adv_type**
Advertising type

esp_ble_addr_type_t **own_addr_type**
Owner bluetooth device address type

esp_bd_addr_t **peer_addr**
Peer device bluetooth device address

esp_ble_addr_type_t **peer_addr_type**
Peer device bluetooth device address type, only support public address type and random address type

esp_ble_adv_channel_t **channel_map**
Advertising channel map

esp_ble_adv_filter_t **adv_filter_policy**
Advertising filter policy

struct esp_ble_adv_data_t
Advertising data content, according to “Supplement to the Bluetooth Core Specification” .

Public Members

bool **set_scan_rsp**
Set this advertising data as scan response or not

bool **include_name**
Advertising data include device name or not

bool **include_txpower**
Advertising data include TX power

int **min_interval**
Advertising data show slave preferred connection min interval. The connection interval in the following manner: $\text{connIntervalmin} = \text{Conn_Interval_Min} * 1.25 \text{ ms}$ **Conn_Interval_Min** range: 0x0006 to 0x0C80 Value of 0xFFFF indicates no specific minimum. Values not defined above are reserved for future use.

int **max_interval**
Advertising data show slave preferred connection max interval. The connection interval in the following manner: $\text{connIntervalmax} = \text{Conn_Interval_Max} * 1.25 \text{ ms}$ **Conn_Interval_Max** range: 0x0006 to 0x0C80 **Conn_Interval_Max** shall be equal to or greater than the **Conn_Interval_Min**. Value of 0xFFFF indicates no specific maximum. Values not defined above are reserved for future use.

int **appearance**
External appearance of device

uint16_t **manufacturer_len**
Manufacturer data length

uint8_t ***p_manufacturer_data**
Manufacturer data point

uint16_t **service_data_len**
Service data length

uint8_t ***p_service_data**
Service data point

uint16_t **service_uuid_len**
Service uuid length

uint8_t ***p_service_uuid**
Service uuid array point

uint8_t **flag**
Advertising flag of discovery mode, see BLE_ADV_DATA_FLAG detail

struct esp_ble_scan_params_t

Ble scan parameters.

Public Members*esp_ble_scan_type_t* **scan_type**

Scan type

esp_ble_addr_type_t **own_addr_type**

Owner address type

esp_ble_scan_filter_t **scan_filter_policy**

Scan filter policy

uint16_t **scan_interval**

Scan interval. This is defined as the time interval from when the Controller started its last LE scan until it begins the subsequent LE scan. Range: 0x0004 to 0x4000 Default: 0x0010 (10 ms) Time = N * 0.625 msec Time Range: 2.5 msec to 10.24 seconds

uint16_t **scan_window**

Scan window. The duration of the LE scan. LE_Scan_Window shall be less than or equal to LE_Scan_Interval Range: 0x0004 to 0x4000 Default: 0x0010 (10 ms) Time = N * 0.625 msec Time Range: 2.5 msec to 10240 msec

esp_ble_scan_duplicate_t **scan_duplicate**

The Scan_Duplicates parameter controls whether the Link Layer should filter out duplicate advertising reports (BLE_SCAN_DUPLICATE_ENABLE) to the Host, or if the Link Layer should generate advertising reports for each packet received

struct esp_gap_conn_params_t

connection parameters information

Public Membersuint16_t **interval**

connection interval

uint16_t **latency**

Slave latency for the connection in number of connection events. Range: 0x0000 to 0x01F3

uint16_t **timeout**

Supervision timeout for the LE Link. Range: 0x000A to 0x0C80. Mandatory Range: 0x000A to 0x0C80 Time = N * 10 msec Time Range: 100 msec to 32 seconds

struct esp_ble_conn_update_params_t

Connection update parameters.

Public Members*esp_bd_addr_t* **bda**

Bluetooth device address

uint16_t **min_int**

Min connection interval

uint16_t **max_int**

Max connection interval

uint16_t **latency**

Slave latency for the connection in number of connection events. Range: 0x0000 to 0x01F3

uint16_t timeout

Supervision timeout for the LE Link. Range: 0x000A to 0x0C80. Mandatory Range: 0x000A to 0x0C80
Time = N * 10 msec Time Range: 100 msec to 32 seconds

struct esp_ble_pkt_data_length_params_t

BLE pkt data length keys.

Public Members

uint16_t rx_len

pkt rx data length value

uint16_t tx_len

pkt tx data length value

struct esp_ble_penc_keys_t

BLE encryption keys.

Public Members

esp_bt_octet16_t ltk

The long term key

esp_bt_octet8_t rand

The random number

uint16_t ediv

The ediv value

uint8_t sec_level

The security level of the security link

uint8_t key_size

The key size(7-16) of the security link

struct esp_ble_pcsrkeys_t

BLE CSRK keys.

Public Members

uint32_t counter

The counter

esp_bt_octet16_t csrkey

The csrkey

uint8_t sec_level

The security level

struct esp_ble_pidkeys_t

BLE pid keys.

Public Members

esp_bt_octet16_t irk

The irk value

esp_ble_addr_type_t addr_type

The address type

esp_bd_addr_t static_addr

The static address

struct esp_ble_lenc_keys_t
BLE Encryption reproduction keys.

Public Members

esp_bt_octet16_t **ltk**
The long term key

uint16_t **div**
The div value

uint8_t **key_size**
The key size of the security link

uint8_t **sec_level**
The security level of the security link

struct esp_ble_lcsrk_keys
BLE SRK keys.

Public Members

uint32_t **counter**
The counter value

uint16_t **div**
The div value

uint8_t **sec_level**
The security level of the security link

esp_bt_octet16_t **csrk**
The csrk key value

struct esp_ble_sec_key_notif_t
Structure associated with ESP_KEY_NOTIF_EVT.

Public Members

esp_bd_addr_t **bd_addr**
peer address

uint32_t **passkey**
the numeric value for comparison. If just_works, do not show this number to UI

struct esp_ble_sec_req_t
Structure of the security request.

Public Members

esp_bd_addr_t **bd_addr**
peer address

struct esp_ble_bond_key_info_t
struct type of the bond key information value

Public Members

esp_ble_key_mask_t **key_mask**
the key mask to indicate witch key is present

esp_ble_penc_keys_t **penc_key**
received peer encryption key

esp_ble_pcsrkeys_t **pcsrk_key**
received peer device SRK

esp_ble_pid_keys_t **pid_key**
peer device ID key

struct esp_ble_bond_dev_t
struct type of the bond device value

Public Members

esp_bd_addr_t **bd_addr**
peer address

esp_ble_bond_key_info_t **bond_key**
the bond key information

struct esp_ble_key_t
union type of the security key value

Public Members

esp_bd_addr_t **bd_addr**
peer address

esp_ble_key_type_t **key_type**
key type of the security link

esp_ble_key_value_t **p_key_value**
the pointer to the key value

struct esp_ble_local_id_keys_t
structure type of the ble local id keys value

Public Members

esp_bt_octet16_t **ir**
the 16 bits of the ir value

esp_bt_octet16_t **irk**
the 16 bits of the ir key value

esp_bt_octet16_t **dhk**
the 16 bits of the dh key value

struct esp_ble_auth_cmpl_t
Structure associated with ESP_AUTH_CMPL_EVT.

Public Members

esp_bd_addr_t **bd_addr**
BD address peer device.

bool **key_present**
Valid link key value in key element

esp_link_key **key**
Link key associated with peer device.

uint8_t **key_type**
The type of Link Key

bool **success**
TRUE of authentication succeeded, FALSE if failed.

uint8_t **fail_reason**
The HCI reason/error code for when success=FALSE

esp_ble_addr_type_t **addr_type**
Peer device address type

esp_bt_dev_type_t **dev_type**
Device type

esp_ble_auth_req_t **auth_mode**
authentication mode

Macros

ESP_BLE_ADV_FLAG_LIMIT_DISC
BLE_ADV_DATA_FLAG data flag bit definition used for advertising data flag

ESP_BLE_ADV_FLAG_GEN_DISC

ESP_BLE_ADV_FLAG_BREDR_NOT_SPT

ESP_BLE_ADV_FLAG_DMT_CONTROLLER_SPT

ESP_BLE_ADV_FLAG_DMT_HOST_SPT

ESP_BLE_ADV_FLAG_NON_LIMIT_DISC

ESP_LE_KEY_NONE

ESP_LE_KEY_PENC

ESP_LE_KEY_PID

ESP_LE_KEY_PCSRK

ESP_LE_KEY_PLK

ESP_LE_KEY_LLK

ESP_LE_KEY_LENC

ESP_LE_KEY_LID

ESP_LE_KEY_LCSRK

ESP_LE_AUTH_NO_BOND

ESP_LE_AUTH_BOND

ESP_LE_AUTH_REQ_MITM

ESP_LE_AUTH_REQ_BOND_MITM
0101

ESP_LE_AUTH_REQ_SC_ONLY

ESP_LE_AUTH_REQ_SC_BOND

ESP_LE_AUTH_REQ_SC_MITM

ESP_LE_AUTH_REQ_SC_MITM_BOND

ESP_BLE_ONLY_ACCEPT_SPECIFIED_AUTH_DISABLE
ESP_BLE_ONLY_ACCEPT_SPECIFIED_AUTH_ENABLE
ESP_BLE_OOB_DISABLE
ESP_BLE_OOB_ENABLE
ESP_IO_CAP_OUT
ESP_IO_CAP_IO
ESP_IO_CAP_IN
ESP_IO_CAP_NONE
ESP_IO_CAP_KBDISP
ESP_BLE_APPEARANCE_UNKNOWN
ESP_BLE_APPEARANCE_GENERIC_PHONE
ESP_BLE_APPEARANCE_GENERIC_COMPUTER
ESP_BLE_APPEARANCE_GENERIC_WATCH
ESP_BLE_APPEARANCE_SPORTS_WATCH
ESP_BLE_APPEARANCE_GENERIC_CLOCK
ESP_BLE_APPEARANCE_GENERIC_DISPLAY
ESP_BLE_APPEARANCE_GENERIC_REMOTE
ESP_BLE_APPEARANCE_GENERIC_EYEGLASSES
ESP_BLE_APPEARANCE_GENERIC_TAG
ESP_BLE_APPEARANCE_GENERIC_KEYRING
ESP_BLE_APPEARANCE_GENERIC_MEDIA_PLAYER
ESP_BLE_APPEARANCE_GENERIC_BARCODE_SCANNER
ESP_BLE_APPEARANCE_GENERIC_THERMOMETER
ESP_BLE_APPEARANCE_THERMOMETER_EAR
ESP_BLE_APPEARANCE_GENERIC_HEART_RATE
ESP_BLE_APPEARANCE_HEART_RATE_BELT
ESP_BLE_APPEARANCE_GENERIC_BLOOD_PRESSURE
ESP_BLE_APPEARANCE_BLOOD_PRESSURE_ARM
ESP_BLE_APPEARANCE_BLOOD_PRESSURE_WRIST
ESP_BLE_APPEARANCE_GENERIC_HID
ESP_BLE_APPEARANCE_HID_KEYBOARD
ESP_BLE_APPEARANCE_HID_MOUSE
ESP_BLE_APPEARANCE_HID_JOYSTICK
ESP_BLE_APPEARANCE_HID_GAMEPAD
ESP_BLE_APPEARANCE_HID_DIGITIZER_TABLET
ESP_BLE_APPEARANCE_HID_CARD_READER
ESP_BLE_APPEARANCE_HID_DIGITAL_PEN
ESP_BLE_APPEARANCE_HID_BARCODE_SCANNER
ESP_BLE_APPEARANCE_GENERIC_GLUCOSE

ESP_BLE_APPEARANCE_GENERIC_WALKING
ESP_BLE_APPEARANCE_WALKING_IN_SHOE
ESP_BLE_APPEARANCE_WALKING_ON_SHOE
ESP_BLE_APPEARANCE_WALKING_ON_HIP
ESP_BLE_APPEARANCE_GENERIC_CYCLING
ESP_BLE_APPEARANCE_CYCLING_COMPUTER
ESP_BLE_APPEARANCE_CYCLING_SPEED
ESP_BLE_APPEARANCE_CYCLING_CADENCE
ESP_BLE_APPEARANCE_CYCLING_POWER
ESP_BLE_APPEARANCE_CYCLING_SPEED_CADENCE
ESP_BLE_APPEARANCE_GENERIC_PULSE_OXIMETER
ESP_BLE_APPEARANCE_PULSE_OXIMETER_FINGERTIP
ESP_BLE_APPEARANCE_PULSE_OXIMETER_WRIST
ESP_BLE_APPEARANCE_GENERIC_WEIGHT
ESP_BLE_APPEARANCE_GENERIC_PERSONAL_MOBILITY_DEVICE
ESP_BLE_APPEARANCE_POWERED_WHEELCHAIR
ESP_BLE_APPEARANCE_MOBILITY_SCOOTER
ESP_BLE_APPEARANCE_GENERIC_CONTINUOUS_GLUCOSE_MONITOR
ESP_BLE_APPEARANCE_GENERIC_INSULIN_PUMP
ESP_BLE_APPEARANCE_INSULIN_PUMP_DURABLE_PUMP
ESP_BLE_APPEARANCE_INSULIN_PUMP_PATCH_PUMP
ESP_BLE_APPEARANCE_INSULIN_PEN
ESP_BLE_APPEARANCE_GENERIC_MEDICATION_DELIVERY
ESP_BLE_APPEARANCE_GENERIC_OUTDOOR_SPORTS
ESP_BLE_APPEARANCE_OUTDOOR_SPORTS_LOCATION
ESP_BLE_APPEARANCE_OUTDOOR_SPORTS_LOCATION_AND_NAV
ESP_BLE_APPEARANCE_OUTDOOR_SPORTS_LOCATION_POD
ESP_BLE_APPEARANCE_OUTDOOR_SPORTS_LOCATION_POD_AND_NAV
ESP_GAP_BLE_CHANNELS_LEN
ESP_GAP_BLE_ADD_WHITELIST_COMPLETE_EVT
 This is the old name, just for backwards compatibility.
ESP_BLE_ADV_DATA_LEN_MAX
 Advertising data maximum length.
ESP_BLE_SCAN_RSP_DATA_LEN_MAX
 Scan response data maximum length.
BLE_BIT (n)

Type Definitions

```
typedef uint8_t esp_ble_key_type_t
typedef uint8_t esp_ble_auth_req_t
    combination of the above bit pattern

typedef uint8_t esp_ble_io_cap_t
    combination of the io capability

typedef uint8_t esp_gap_ble_channels[ESP_GAP_BLE_CHANNELS_LEN]
typedef uint8_t esp_duplicate_info_t[ESP_BD_ADDR_LEN]
typedef void (*esp_gap_ble_cb_t) (esp_gap_ble_cb_event_t event, esp_ble_gap_cb_param_t
    *param)
    GAP callback function type.
```

Parameters

- event: : Event type
- param: : Point to callback parameter, currently is union type

Enumerations

```
enum esp_gap_ble_cb_event_t
    GAP BLE callback event type.
```

Values:

```
ESP_GAP_BLE_ADV_DATA_SET_COMPLETE_EVT = 0
    When advertising data set complete, the event comes

ESP_GAP_BLE_SCAN_RSP_DATA_SET_COMPLETE_EVT
    When scan response data set complete, the event comes

ESP_GAP_BLE_SCAN_PARAM_SET_COMPLETE_EVT
    When scan parameters set complete, the event comes

ESP_GAP_BLE_SCAN_RESULT_EVT
    When one scan result ready, the event comes each time

ESP_GAP_BLE_ADV_DATA_RAW_SET_COMPLETE_EVT
    When raw advertising data set complete, the event comes

ESP_GAP_BLE_SCAN_RSP_DATA_RAW_SET_COMPLETE_EVT
    When raw advertising data set complete, the event comes

ESP_GAP_BLE_ADV_START_COMPLETE_EVT
    When starting advertising complete, the event comes

ESP_GAP_BLE_SCAN_START_COMPLETE_EVT
    When starting scan complete, the event comes

ESP_GAP_BLE_AUTH_CMPL_EVT
    Authentication complete indication.

ESP_GAP_BLE_KEY_EVT
    BLE key event for peer device keys

ESP_GAP_BLE_SEC_REQ_EVT
    BLE security request

ESP_GAP_BLE_PASSKEY_NOTIF_EVT
    passkey notification event

ESP_GAP_BLE_PASSKEY_REQ_EVT
    passkey request event

ESP_GAP_BLE_OOB_REQ_EVT
    OOB request event
```

ESP_GAP_BLE_LOCAL_IR_EVT

BLE local IR event

ESP_GAP_BLE_LOCAL_ER_EVT

BLE local ER event

ESP_GAP_BLE_NC_REQ_EVT

Numeric Comparison request event

ESP_GAP_BLE_ADV_STOP_COMPLETE_EVT

When stopping adv complete, the event comes

ESP_GAP_BLE_SCAN_STOP_COMPLETE_EVT

When stopping scan complete, the event comes

ESP_GAP_BLE_SET_STATIC_RAND_ADDR_EVT

When setting the static rand address complete, the event comes

ESP_GAP_BLE_UPDATE_CONN_PARAMS_EVT

When updating connection parameters complete, the event comes

ESP_GAP_BLE_SET_PKT_LENGTH_COMPLETE_EVT

When setting pkt length complete, the event comes

ESP_GAP_BLE_SET_LOCAL_PRIVACY_COMPLETE_EVT

When enabling/disabling privacy on the local device complete, the event comes

ESP_GAP_BLE_REMOVE_BOND_DEV_COMPLETE_EVT

When removing the bond device complete, the event comes

ESP_GAP_BLE_CLEAR_BOND_DEV_COMPLETE_EVT

When clearing the bond device clear complete, the event comes

ESP_GAP_BLE_GET_BOND_DEV_COMPLETE_EVT

When getting the bond device list complete, the event comes

ESP_GAP_BLE_READ_RSSI_COMPLETE_EVT

When reading the rssi complete, the event comes

ESP_GAP_BLE_UPDATE_WHITELIST_COMPLETE_EVT

When adding or removing whitelist complete, the event comes

ESP_GAP_BLE_UPDATE_DUPLICATE_EXCEPTIONAL_LIST_COMPLETE_EVT

When updating duplicate exceptional list complete, the event comes

ESP_GAP_BLE_SET_CHANNELS_EVT

When setting BLE channels complete, the event comes

ESP_GAP_BLE_EVT_MAX

enum esp_ble_adv_data_type

The type of advertising data(not adv_type)

Values:

ESP_BLE_AD_TYPE_FLAG = 0x01

ESP_BLE_AD_TYPE_16SRV_PART = 0x02

ESP_BLE_AD_TYPE_16SRV_CMPL = 0x03

ESP_BLE_AD_TYPE_32SRV_PART = 0x04

ESP_BLE_AD_TYPE_32SRV_CMPL = 0x05

ESP_BLE_AD_TYPE_128SRV_PART = 0x06

ESP_BLE_AD_TYPE_128SRV_CMPL = 0x07

ESP_BLE_AD_TYPE_NAME_SHORT = 0x08

ESP_BLE_AD_TYPE_NAME_CMPL = 0x09


```
ESP_BLE_AD_TYPE_TX_PWR = 0x0A
ESP_BLE_AD_TYPE_DEV_CLASS = 0x0D
ESP_BLE_AD_TYPE_SM_TK = 0x10
ESP_BLE_AD_TYPE_SM_OOB_FLAG = 0x11
ESP_BLE_AD_TYPE_INT_RANGE = 0x12
ESP_BLE_AD_TYPE_SOL_SRV_UUID = 0x14
ESP_BLE_AD_TYPE_128SOL_SRV_UUID = 0x15
ESP_BLE_AD_TYPE_SERVICE_DATA = 0x16
ESP_BLE_AD_TYPE_PUBLIC_TARGET = 0x17
ESP_BLE_AD_TYPE_RANDOM_TARGET = 0x18
ESP_BLE_AD_TYPE_APPEARANCE = 0x19
ESP_BLE_AD_TYPE_ADV_INT = 0x1A
ESP_BLE_AD_TYPE_LE_DEV_ADDR = 0x1b
ESP_BLE_AD_TYPE_LE_ROLE = 0x1c
ESP_BLE_AD_TYPE_SPAIR_C256 = 0x1d
ESP_BLE_AD_TYPE_SPAIR_R256 = 0x1e
ESP_BLE_AD_TYPE_32SOL_SRV_UUID = 0x1f
ESP_BLE_AD_TYPE_32SERVICE_DATA = 0x20
ESP_BLE_AD_TYPE_128SERVICE_DATA = 0x21
ESP_BLE_AD_TYPE_LE_SECURE_CONFIRM = 0x22
ESP_BLE_AD_TYPE_LE_SECURE_RANDOM = 0x23
ESP_BLE_AD_TYPE_URI = 0x24
ESP_BLE_AD_TYPE_INDOOR_POSITION = 0x25
ESP_BLE_AD_TYPE_TRANS_DISC_DATA = 0x26
ESP_BLE_AD_TYPE_LE_SUPPORT_FEATURE = 0x27
ESP_BLE_AD_TYPE_CHAN_MAP_UPDATE = 0x28
ESP_BLE_AD_MANUFACTURER_SPECIFIC_TYPE = 0xFF
```

```
enum esp_ble_adv_type_t
```

Advertising mode.

Values:

```
ADV_TYPE_IND = 0x00
ADV_TYPE_DIRECT_IND_HIGH = 0x01
ADV_TYPE_SCAN_IND = 0x02
ADV_TYPE_NONCONN_IND = 0x03
ADV_TYPE_DIRECT_IND_LOW = 0x04
```

```
enum esp_ble_adv_channel_t
```

Advertising channel mask.

Values:

```
ADV_CHNL_37 = 0x01
ADV_CHNL_38 = 0x02
```

ADV_CHNL_39 = 0x04

ADV_CHNL_ALL = 0x07

enum esp_ble_adv_filter_t

Values:

ADV_FILTER_ALLOW_SCAN_ANY_CON_ANY = 0x00

Allow both scan and connection requests from anyone.

ADV_FILTER_ALLOW_SCAN_WLST_CON_ANY

Allow both scan req from White List devices only and connection req from anyone.

ADV_FILTER_ALLOW_SCAN_ANY_CON_WLST

Allow both scan req from anyone and connection req from White List devices only.

ADV_FILTER_ALLOW_SCAN_WLST_CON_WLST

Allow scan and connection requests from White List devices only.

enum esp_ble_sec_act_t

Values:

ESP_BLE_SEC_ENCRYPT = 1

ESP_BLE_SEC_ENCRYPT_NO_MITM

ESP_BLE_SEC_ENCRYPT_MITM

enum esp_ble_sm_param_t

Values:

ESP_BLE_SM_PASSKEY = 0

ESP_BLE_SM_AUTHEN_REQ_MODE

ESP_BLE_SM_IOCAP_MODE

ESP_BLE_SM_SET_INIT_KEY

ESP_BLE_SM_SET_RSP_KEY

ESP_BLE_SM_MAX_KEY_SIZE

ESP_BLE_SM_MIN_KEY_SIZE

ESP_BLE_SM_SET_STATIC_PASSKEY

ESP_BLE_SM_CLEAR_STATIC_PASSKEY

ESP_BLE_SM_ONLY_ACCEPT_SPECIFIED_SEC_AUTH

ESP_BLE_SM_OOB_SUPPORT

ESP_BLE_APP_ENC_KEY_SIZE

ESP_BLE_SM_MAX_PARAM

enum esp_ble_scan_type_t

Ble scan type.

Values:

BLE_SCAN_TYPE_PASSIVE = 0x0

Passive scan

BLE_SCAN_TYPE_ACTIVE = 0x1

Active scan

enum esp_ble_scan_filter_t

Ble scan filter type.

Values:

BLE_SCAN_FILTER_ALLOW_ALL = 0x0

Accept all :

1. advertisement packets except directed advertising packets not addressed to this device (default).

BLE_SCAN_FILTER_ALLOW_ONLY_WLST = 0x1

Accept only :

1. advertisement packets from devices where the advertiser' s address is in the White list.
2. Directed advertising packets which are not addressed for this device shall be ignored.

BLE_SCAN_FILTER_ALLOW_UND_RPA_DIR = 0x2

Accept all :

1. undirected advertisement packets, and
2. directed advertising packets where the initiator address is a resolvable private address, and
3. directed advertising packets addressed to this device.

BLE_SCAN_FILTER_ALLOW_WLIST_PRA_DIR = 0x3

Accept all :

1. advertisement packets from devices where the advertiser' s address is in the White list, and
2. directed advertising packets where the initiator address is a resolvable private address, and
3. directed advertising packets addressed to this device.

enum esp_ble_scan_duplicate_t

Ble scan duplicate type.

Values:

BLE_SCAN_DUPLICATE_DISABLE = 0x0

the Link Layer should generate advertising reports to the host for each packet received

BLE_SCAN_DUPLICATE_ENABLE = 0x1

the Link Layer should filter out duplicate advertising reports to the Host

BLE_SCAN_DUPLICATE_MAX = 0x2

0x02 –0xFF, Reserved for future use

enum esp_gap_search_evt_t

Sub Event of ESP_GAP_BLE_SCAN_RESULT_EVT.

Values:

ESP_GAP_SEARCH_INQ_RES_EVT = 0

Inquiry result for a peer device.

ESP_GAP_SEARCH_INQ_CMPL_EVT = 1

Inquiry complete.

ESP_GAP_SEARCH_DISC_RES_EVT = 2

Discovery result for a peer device.

ESP_GAP_SEARCH_DISC_BLE_RES_EVT = 3

Discovery result for BLE GATT based service on a peer device.

ESP_GAP_SEARCH_DISC_CMPL_EVT = 4

Discovery complete.

ESP_GAP_SEARCH_DI_DISC_CMPL_EVT = 5

Discovery complete.

ESP_GAP_SEARCH_SEARCH_CANCEL_CMPL_EVT = 6

Search cancelled

ESP_GAP_SEARCH_INQ_DISCARD_NUM_EVT = 7

The number of pkt discarded by flow control

enum esp_ble_evt_type_t

Ble scan result event type, to indicate the result is scan response or advertising data or other.

Values:

- ESP_BLE_EVT_CONN_ADV** = 0x00
Connectable undirected advertising (ADV_IND)
- ESP_BLE_EVT_CONN_DIR_ADV** = 0x01
Connectable directed advertising (ADV_DIRECT_IND)
- ESP_BLE_EVT_DISC_ADV** = 0x02
Scannable undirected advertising (ADV_SCAN_IND)
- ESP_BLE_EVT_NON_CONN_ADV** = 0x03
Non connectable undirected advertising (ADV_NONCONN_IND)
- ESP_BLE_EVT_SCAN_RSP** = 0x04
Scan Response (SCAN_RSP)

enum esp_ble_wl_opration_t

Values:

- ESP_BLE_WHITELIST_REMOVE** = 0X00
remove mac from whitelist
- ESP_BLE_WHITELIST_ADD** = 0X01
add address to whitelist

enum esp_bt_duplicate_exceptional_subcode_type_t

Values:

- ESP_BLE_DUPLICATE_EXCEPTIONAL_LIST_ADD** = 0
Add device info into duplicate scan exceptional list
- ESP_BLE_DUPLICATE_EXCEPTIONAL_LIST_REMOVE**
Remove device info from duplicate scan exceptional list
- ESP_BLE_DUPLICATE_EXCEPTIONAL_LIST_CLEAN**
Clean duplicate scan exceptional list

enum esp_ble_duplicate_exceptional_info_type_t

Values:

- ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_INFO_ADV_ADDR** = 0
BLE advertising address, device info will be added into ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_ADDR_LIST
- ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_INFO_MESH_LINK_ID**
BLE mesh link ID, it is for BLE mesh, device info will be added into ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_MESH_LINK_ID_LIST
- ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_INFO_MESH_BEACON_TYPE**
BLE mesh beacon AD type, the format is | Len | 0x2B | Beacon Type | Beacon Data |
- ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_INFO_MESH_PROV_SRV_ADV**
BLE mesh provisioning service uuid, the format is | 0x02 | 0x01 | flags | 0x03 | 0x03 | 0x1827 | ... |
- ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_INFO_MESH_PROXY_SRV_ADV**
BLE mesh adv with proxy service uuid, the format is | 0x02 | 0x01 | flags | 0x03 | 0x03 | 0x1828 | ... |

enum esp_duplicate_scan_exceptional_list_type_t

Values:

- ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_ADDR_LIST** = BLE_BIT(0)
duplicate scan exceptional addr list
- ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_MESH_LINK_ID_LIST** = BLE_BIT(1)
duplicate scan exceptional mesh link ID list

ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_MESH_BEACON_TYPE_LIST = BLE_BIT(2)
duplicate scan exceptional mesh beacon type list

ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_MESH_PROV_SRV_ADV_LIST = BLE_BIT(3)
duplicate scan exceptional mesh adv with provisioning service uuid

ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_MESH_PROXY_SRV_ADV_LIST = BLE_BIT(4)
duplicate scan exceptional mesh adv with provisioning service uuid

ESP_BLE_DUPLICATE_SCAN_EXCEPTIONAL_ALL_LIST = 0xFFFF
duplicate scan exceptional all list

GATT DEFINES

[Overview](#) [Instructions](#)

[Application Example](#) [Instructions](#)

API Reference

Header File

- [bt/host/bluedroid/api/include/api/esp_gatt_defs.h](#)

Unions

union esp_gatt_rsp_t
#include <esp_gatt_defs.h> GATT remote read request response type.

Public Members

esp_gatt_value_t **attr_value**
Gatt attribute structure

uint16_t **handle**
Gatt attribute handle

Structures

struct esp_gatt_id_t
Gatt id, include uuid and instance id.

Public Members

esp_bt_uuid_t **uuid**
UUID

uint8_t **inst_id**
Instance id

struct esp_gatt_srvc_id_t
Gatt service id, include id (uuid and instance id) and primary flag.

Public Members

esp_gatt_id_t **id**
Gatt id, include uuid and instance

bool **is_primary**
This service is primary or not

struct esp_attr_desc_t
Attribute description (used to create database)

Public Members

uint16_t **uuid_length**
UUID length

uint8_t ***uuid_p**
UUID value

uint16_t **perm**
Attribute permission

uint16_t **max_length**
Maximum length of the element

uint16_t **length**
Current length of the element

uint8_t ***value**
Element value array

struct esp_attr_control_t
attribute auto response flag

Public Members

uint8_t **auto_rsp**
if auto_rsp set to ESP_GATT_RSP_BY_APP, means the response of Write/Read operation will be replied by application. if auto_rsp set to ESP_GATT_AUTO_RSP, means the response of Write/Read operation will be replied by GATT stack automatically.

struct esp_gatts_attr_db_t
attribute type added to the gatt server database

Public Members

esp_attr_control_t **attr_control**
The attribute control type

esp_attr_desc_t **att_desc**
The attribute type

struct esp_attr_value_t
set the attribute value type

Public Members

uint16_t **attr_max_len**
attribute max value length

`uint16_t attr_len`
attribute current value length

`uint8_t *attr_value`
the pointer to attribute value

struct esp_gatts_incl_svc_desc_t
Gatt include service entry element.

Public Members

`uint16_t start_hdl`
Gatt start handle value of included service

`uint16_t end_hdl`
Gatt end handle value of included service

`uint16_t uuid`
Gatt attribute value UUID of included service

struct esp_gatts_incl128_svc_desc_t
Gatt include 128 bit service entry element.

Public Members

`uint16_t start_hdl`
Gatt start handle value of included 128 bit service

`uint16_t end_hdl`
Gatt end handle value of included 128 bit service

struct esp_gatt_value_t
Gatt attribute value.

Public Members

`uint8_t value[ESP_GATT_MAX_ATTR_LEN]`
Gatt attribute value

`uint16_t handle`
Gatt attribute handle

`uint16_t offset`
Gatt attribute value offset

`uint16_t len`
Gatt attribute value length

`uint8_t auth_req`
Gatt authentication request

struct esp_gatt_conn_params_t
Connection parameters information.

Public Members

`uint16_t interval`
connection interval

`uint16_t latency`
Slave latency for the connection in number of connection events. Range: 0x0000 to 0x01F3

uint16_t timeout

Supervision timeout for the LE Link. Range: 0x000A to 0x0C80. Mandatory Range: 0x000A to 0x0C80
Time = N * 10 msec Time Range: 100 msec to 32 seconds

struct esp_gattc_multi_t
read multiple attribute

Public Members

uint8_t num_attr

The number of the attribute

uint16_t handles[ESP_GATT_MAX_READ_MULTI_HANDLES]

The handles list

struct esp_gattc_db_elem_t
data base attribute element

Public Members

esp_gatt_db_attr_type_t **type**

The attribute type

uint16_t attribute_handle

The attribute handle, it' s valid for all of the type

uint16_t start_handle

The service start handle, it' s valid only when the type = ESP_GATT_DB_PRIMARY_SERVICE or ESP_GATT_DB_SECONDARY_SERVICE

uint16_t end_handle

The service end handle, it' s valid only when the type = ESP_GATT_DB_PRIMARY_SERVICE or ESP_GATT_DB_SECONDARY_SERVICE

esp_gatt_char_prop_t **properties**

The characteristic properties, it' s valid only when the type = ESP_GATT_DB_CHARACTERISTIC

esp_bt_uuid_t **uuid**

The attribute uuid, it' s valid for all of the type

struct esp_gattc_service_elem_t
service element

Public Members

bool is_primary

The service flag, true if the service is primary service, else is secondary service

uint16_t start_handle

The start handle of the service

uint16_t end_handle

The end handle of the service

esp_bt_uuid_t **uuid**

The uuid of the service

struct esp_gattc_char_elem_t
characteristic element

Public Members

`uint16_t char_handle`
The characteristic handle

`esp_gatt_char_prop_t properties`
The characteristic properties

`esp_bt_uuid_t uuid`
The characteristic uuid

`struct esp_gattc_descr_elem_t`
descriptor element

Public Members

`uint16_t handle`
The characteristic descriptor handle

`esp_bt_uuid_t uuid`
The characteristic descriptor uuid

`struct esp_gattc_incl_svc_elem_t`
include service element

Public Members

`uint16_t handle`
The include service current attribute handle

`uint16_t incl_srvc_s_handle`
The start handle of the service which has been included

`uint16_t incl_srvc_e_handle`
The end handle of the service which has been included

`esp_bt_uuid_t uuid`
The include service uuid

Macros

`ESP_GATT_UUID_IMMEDIATE_ALERT_SVC`
All “ESP_GATT_UUID_XXX” is attribute types

`ESP_GATT_UUID_LINK_LOSS_SVC`

`ESP_GATT_UUID_TX_POWER_SVC`

`ESP_GATT_UUID_CURRENT_TIME_SVC`

`ESP_GATT_UUID_REF_TIME_UPDATE_SVC`

`ESP_GATT_UUID_NEXT_DST_CHANGE_SVC`

`ESP_GATT_UUID_GLUCOSE_SVC`

`ESP_GATT_UUID_HEALTH_THERMOM_SVC`

`ESP_GATT_UUID_DEVICE_INFO_SVC`

`ESP_GATT_UUID_HEART_RATE_SVC`

`ESP_GATT_UUID_PHONE_ALERT_STATUS_SVC`

`ESP_GATT_UUID_BATTERY_SERVICE_SVC`

`ESP_GATT_UUID_BLOOD_PRESSURE_SVC`

ESP_GATT_UUID_ALERT_NTF_SVC
ESP_GATT_UUID_HID_SVC
ESP_GATT_UUID_SCAN_PARAMETERS_SVC
ESP_GATT_UUID_RUNNING_SPEED_CADENCE_SVC
ESP_GATT_UUID_Automation_IO_SVC
ESP_GATT_UUID_CYCLING_SPEED_CADENCE_SVC
ESP_GATT_UUID_CYCLING_POWER_SVC
ESP_GATT_UUID_LOCATION_AND_NAVIGATION_SVC
ESP_GATT_UUID_ENVIRONMENTAL_SENSING_SVC
ESP_GATT_UUID_BODY_COMPOSITION
ESP_GATT_UUID_USER_DATA_SVC
ESP_GATT_UUID_WEIGHT_SCALE_SVC
ESP_GATT_UUID_BOND_MANAGEMENT_SVC
ESP_GATT_UUID_CONT_GLUCOSE_MONITOR_SVC
ESP_GATT_UUID_PRI_SERVICE
ESP_GATT_UUID_SEC_SERVICE
ESP_GATT_UUID_INCLUDE_SERVICE
ESP_GATT_UUID_CHAR_DECLARE
ESP_GATT_UUID_CHAR_EXT_PROP
ESP_GATT_UUID_CHAR_DESCRIPTION
ESP_GATT_UUID_CHAR_CLIENT_CONFIG
ESP_GATT_UUID_CHAR_SRVR_CONFIG
ESP_GATT_UUID_CHAR_PRESENT_FORMAT
ESP_GATT_UUID_CHAR_AGG_FORMAT
ESP_GATT_UUID_CHAR_VALID_RANGE
ESP_GATT_UUID_EXT_RPT_REF_DESCR
ESP_GATT_UUID_RPT_REF_DESCR
ESP_GATT_UUID_NUM_DIGITALS_DESCR
ESP_GATT_UUID_VALUE_TRIGGER_DESCR
ESP_GATT_UUID_ENV_SENSING_CONFIG_DESCR
ESP_GATT_UUID_ENV_SENSING_MEASUREMENT_DESCR
ESP_GATT_UUID_ENV_SENSING_TRIGGER_DESCR
ESP_GATT_UUID_TIME_TRIGGER_DESCR
ESP_GATT_UUID_GAP_DEVICE_NAME
ESP_GATT_UUID_GAP_ICON
ESP_GATT_UUID_GAP_PREF_CONN_PARAM
ESP_GATT_UUID_GAP_CENTRAL_ADDR_RESOL
ESP_GATT_UUID_GATT_SRV_CHGD
ESP_GATT_UUID_ALERT_LEVEL

ESP_GATT_UUID_TX_POWER_LEVEL
ESP_GATT_UUID_CURRENT_TIME
ESP_GATT_UUID_LOCAL_TIME_INFO
ESP_GATT_UUID_REF_TIME_INFO
ESP_GATT_UUID_NW_STATUS
ESP_GATT_UUID_NW_TRIGGER
ESP_GATT_UUID_ALERT_STATUS
ESP_GATT_UUID_RINGER_CP
ESP_GATT_UUID_RINGER_SETTING
ESP_GATT_UUID_GM_MEASUREMENT
ESP_GATT_UUID_GM_CONTEXT
ESP_GATT_UUID_GM_CONTROL_POINT
ESP_GATT_UUID_GM_FEATURE
ESP_GATT_UUID_SYSTEM_ID
ESP_GATT_UUID_MODEL_NUMBER_STR
ESP_GATT_UUID_SERIAL_NUMBER_STR
ESP_GATT_UUID_FW_VERSION_STR
ESP_GATT_UUID_HW_VERSION_STR
ESP_GATT_UUID_SW_VERSION_STR
ESP_GATT_UUID_MANU_NAME
ESP_GATT_UUID_IEEE_DATA
ESP_GATT_UUID_PNP_ID
ESP_GATT_UUID_HID_INFORMATION
ESP_GATT_UUID_HID_REPORT_MAP
ESP_GATT_UUID_HID_CONTROL_POINT
ESP_GATT_UUID_HID_REPORT
ESP_GATT_UUID_HID_PROTO_MODE
ESP_GATT_UUID_HID_BT_KB_INPUT
ESP_GATT_UUID_HID_BT_KB_OUTPUT
ESP_GATT_UUID_HID_BT_MOUSE_INPUT
ESP_GATT_HEART_RATE_MEAS
Heart Rate Measurement.
ESP_GATT_BODY_SENSOR_LOCATION
Body Sensor Location.
ESP_GATT_HEART_RATE_CNTL_POINT
Heart Rate Control Point.
ESP_GATT_UUID_BATTERY_LEVEL
ESP_GATT_UUID_SC_CONTROL_POINT
ESP_GATT_UUID_SENSOR_LOCATION
ESP_GATT_UUID_RSC_MEASUREMENT

ESP_GATT_UUID_RSC_FEATURE

ESP_GATT_UUID_CSC_MEASUREMENT

ESP_GATT_UUID_CSC_FEATURE

ESP_GATT_UUID_SCAN_INT_WINDOW

ESP_GATT_UUID_SCAN_REFRESH

ESP_GATT_ILLEGAL_UUID

GATT INVALID UUID.

ESP_GATT_ILLEGAL_HANDLE

GATT INVALID HANDLE.

ESP_GATT_ATTR_HANDLE_MAX

GATT attribute max handle.

ESP_GATT_MAX_READ_MULTI_HANDLES

ESP_GATT_PERM_READ

Attribute permissions.

ESP_GATT_PERM_READ_ENCRYPTED

ESP_GATT_PERM_READ_ENC_MITM

ESP_GATT_PERM_WRITE

ESP_GATT_PERM_WRITE_ENCRYPTED

ESP_GATT_PERM_WRITE_ENC_MITM

ESP_GATT_PERM_WRITE_SIGNED

ESP_GATT_PERM_WRITE_SIGNED_MITM

ESP_GATT_PERM_READ_AUTHORIZATION

ESP_GATT_PERM_WRITE_AUTHORIZATION

ESP_GATT_CHAR_PROP_BIT_BROADCAST

ESP_GATT_CHAR_PROP_BIT_READ

ESP_GATT_CHAR_PROP_BIT_WRITE_NR

ESP_GATT_CHAR_PROP_BIT_WRITE

ESP_GATT_CHAR_PROP_BIT_NOTIFY

ESP_GATT_CHAR_PROP_BIT_INDICATE

ESP_GATT_CHAR_PROP_BIT_AUTH

ESP_GATT_CHAR_PROP_BIT_EXT_PROP

ESP_GATT_MAX_ATTR_LEN

GATT maximum attribute length.

ESP_GATT_RSP_BY_APP

ESP_GATT_AUTO_RSP

ESP_GATT_IF_NONE

If callback report `gattc_if/gatts_if` as this macro, means this event is not correspond to any app

Type Definitions

```
typedef uint16_t esp_gatt_perm_t
```

```
typedef uint8_t esp_gatt_char_prop_t
```

```
typedef uint8_t esp_gatt_if_t
```

Gatt interface type, different application on GATT client use different gatt_if

Enumerations

```
enum esp_gatt_prep_write_type
```

Attribute write data type from the client.

Values:

```
ESP_GATT_PREP_WRITE_CANCEL = 0x00
```

Prepare write cancel

```
ESP_GATT_PREP_WRITE_EXEC = 0x01
```

Prepare write execute

```
enum esp_gatt_status_t
```

GATT success code and error codes.

Values:

```
ESP_GATT_OK = 0x0
```

```
ESP_GATT_INVALID_HANDLE = 0x01
```

```
ESP_GATT_READ_NOT_PERMIT = 0x02
```

```
ESP_GATT_WRITE_NOT_PERMIT = 0x03
```

```
ESP_GATT_INVALID_PDU = 0x04
```

```
ESP_GATT_INSUF_AUTHENTICATION = 0x05
```

```
ESP_GATT_REQ_NOT_SUPPORTED = 0x06
```

```
ESP_GATT_INVALID_OFFSET = 0x07
```

```
ESP_GATT_INSUF_AUTHORIZATION = 0x08
```

```
ESP_GATT_PREPARE_Q_FULL = 0x09
```

```
ESP_GATT_NOT_FOUND = 0x0a
```

```
ESP_GATT_NOT_LONG = 0x0b
```

```
ESP_GATT_INSUF_KEY_SIZE = 0x0c
```

```
ESP_GATT_INVALID_ATTR_LEN = 0x0d
```

```
ESP_GATT_ERR_UNLIKELY = 0x0e
```

```
ESP_GATT_INSUF_ENCRYPTION = 0x0f
```

```
ESP_GATT_UNSUPPORT_GRP_TYPE = 0x10
```

```
ESP_GATT_INSUF_RESOURCE = 0x11
```

```
ESP_GATT_NO_RESOURCES = 0x80
```

```
ESP_GATT_INTERNAL_ERROR = 0x81
```

```
ESP_GATT_WRONG_STATE = 0x82
```

```
ESP_GATT_DB_FULL = 0x83
```

```
ESP_GATT_BUSY = 0x84
```

```
ESP_GATT_ERROR = 0x85
```

```
ESP_GATT_CMD_STARTED = 0x86
```

ESP_GATT_ILLEGAL_PARAMETER = 0x87
ESP_GATT_PENDING = 0x88
ESP_GATT_AUTH_FAIL = 0x89
ESP_GATT_MORE = 0x8a
ESP_GATT_INVALID_CFG = 0x8b
ESP_GATT_SERVICE_STARTED = 0x8c
ESP_GATT_ENCRYPED_MITM = *ESP_GATT_OK*
ESP_GATT_ENCRYPED_NO_MITM = 0x8d
ESP_GATT_NOT_ENCRYPTED = 0x8e
ESP_GATT_CONGESTED = 0x8f
ESP_GATT_DUP_REG = 0x90
ESP_GATT_ALREADY_OPEN = 0x91
ESP_GATT_CANCEL = 0x92
ESP_GATT_STACK_RSP = 0xe0
ESP_GATT_APP_RSP = 0xe1
ESP_GATT_UNKNOWN_ERROR = 0xef
ESP_GATT_CCC_CFG_ERR = 0xfd
ESP_GATT_PRC_IN_PROGRESS = 0xfe
ESP_GATT_OUT_OF_RANGE = 0xff

enum esp_gatt_conn_reason_t

Gatt Connection reason enum.

Values:

ESP_GATT_CONN_UNKNOWN = 0
Gatt connection unknown

ESP_GATT_CONN_L2C_FAILURE = 1
General L2cap failure

ESP_GATT_CONN_TIMEOUT = 0x08
Connection timeout

ESP_GATT_CONN_TERMINATE_PEER_USER = 0x13
Connection terminate by peer user

ESP_GATT_CONN_TERMINATE_LOCAL_HOST = 0x16
Connection terminated by local host

ESP_GATT_CONN_FAIL_ESTABLISH = 0x3e
Connection fail to establish

ESP_GATT_CONN_LMP_TIMEOUT = 0x22
Connection fail for LMP response tout

ESP_GATT_CONN_CONN_CANCEL = 0x0100
L2CAP connection cancelled

ESP_GATT_CONN_NONE = 0x0101
No connection to cancel

enum esp_gatt_auth_req_t

Gatt authentication request type.

Values:

```
ESP_GATT_AUTH_REQ_NONE = 0
ESP_GATT_AUTH_REQ_NO_MITM = 1
ESP_GATT_AUTH_REQ_MITM = 2
ESP_GATT_AUTH_REQ_SIGNED_NO_MITM = 3
ESP_GATT_AUTH_REQ_SIGNED_MITM = 4
```

```
enum esp_service_source_t
```

Values:

```
ESP_GATT_SERVICE_FROM_REMOTE_DEVICE = 0
ESP_GATT_SERVICE_FROM_NVS_FLASH = 1
ESP_GATT_SERVICE_FROM_UNKNOWN = 2
```

```
enum esp_gatt_write_type_t
```

Gatt write type.

Values:

```
ESP_GATT_WRITE_TYPE_NO_RSP = 1
    Gatt write attribute need no response
ESP_GATT_WRITE_TYPE_RSP
    Gatt write attribute need remote response
```

```
enum esp_gatt_db_attr_type_t
```

the type of attribute element

Values:

```
ESP_GATT_DB_PRIMARY_SERVICE
    Gattc primary service attribute type in the cache
ESP_GATT_DB_SECONDARY_SERVICE
    Gattc secondary service attribute type in the cache
ESP_GATT_DB_CHARACTERISTIC
    Gattc characteristic attribute type in the cache
ESP_GATT_DB_DESCRIPTOR
    Gattc characteristic descriptor attribute type in the cache
ESP_GATT_DB_INCLUDED_SERVICE
    Gattc include service attribute type in the cache
ESP_GATT_DB_ALL
    Gattc all the attribute (primary service & secondary service & include service & char & descriptor) type
    in the cache
```

GATT SERVER API

Overview [Instructions](#)

Application Example Check [bluetooth/bluedroid/ble](#) folder in ESP-IDF examples, which contains the following demos and their tutorials:

- This is a GATT sever demo and its tutorial. This demo creates a GATT service with an attribute table, which releases the user from adding attributes one by one. This is the recommended method of adding attributes.
 - [bluetooth/bluedroid/ble/gatt_server_service_table](#)
 - [GATT Server Service Table Example Walkthrough](#)
- This is a GATT server demo and its tutorial. This demo creates a GATT service by adding attributes one by one as defined by Bluedroid. The recommended method of adding attributes is presented in example above.

- [bluetooth/bluedroid/ble/gatt_server](#)
- [GATT Server Example Walkthrough](#)
- This is a BLE SPP-Like demo. This demo, which acts as a GATT server, can receive data from UART and then send the data to the peer device automatically.
 - [bluetooth/bluedroid/ble/ble_spp_server](#)

API Reference

Header File

- [bt/host/bluedroid/api/include/api/esp_gatts_api.h](#)

Functions

esp_err_t **esp_ble_gatts_register_callback** (*esp_gatts_cb_t* callback)

This function is called to register application callbacks with BTA GATTS module.

Return

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gatts_app_register** (uint16_t app_id)

This function is called to register application identifier.

Return

- ESP_OK : success
- other : failed

esp_err_t **esp_ble_gatts_app_unregister** (*esp_gatt_if_t* gatts_if)

unregister with GATT Server.

Return

- ESP_OK : success
- other : failed

Parameters

- [in] gatts_if: GATT server access interface

esp_err_t **esp_ble_gatts_create_service** (*esp_gatt_if_t* gatts_if, *esp_gatt_srvc_id_t* *service_id, uint16_t num_handle)

Create a service. When service creation is done, a callback event ESP_GATTS_CREATE_EVT is called to report status and service ID to the profile. The service ID obtained in the callback function needs to be used when adding included service and characteristics/descriptors into the service.

Return

- ESP_OK : success
- other : failed

Parameters

- [in] gatts_if: GATT server access interface
- [in] service_id: service ID.
- [in] num_handle: number of handle requested for this service.

esp_err_t **esp_ble_gatts_create_attr_tab** (const *esp_gatts_attr_db_t* *gatts_attr_db, *esp_gatt_if_t* gatts_if, uint8_t max_nb_attr, uint8_t srvc_inst_id)

Create a service attribute tab.

Return

- ESP_OK : success
- other : failed

Parameters

- [in] gatts_attr_db: the pointer to the service attr tab
- [in] gatts_if: GATT server access interface
- [in] max_nb_attr: the number of attribute to be added to the service database.

- [in] `srvc_inst_id`: the instance id of the service

`esp_err_t esp_ble_gatts_add_included_service` (uint16_t *service_handle*, uint16_t *included_service_handle*)

This function is called to add an included service. This function have to be called between 'esp_ble_gatts_create_service' and 'esp_ble_gatts_add_char'. After included service is included, a callback event ESP_GATTS_ADD_INCL_SRVC_EVT is reported the included service ID.

Return

- ESP_OK : success
- other : failed

Parameters

- [in] `service_handle`: service handle to which this included service is to be added.
- [in] `included_service_handle`: the service ID to be included.

`esp_err_t esp_ble_gatts_add_char` (uint16_t *service_handle*, *esp_bt_uuid_t* **char_uuid*, *esp_gatt_perm_t* *perm*, *esp_gatt_char_prop_t* *property*, *esp_attr_value_t* **char_val*, *esp_attr_control_t* **control*)

This function is called to add a characteristic into a service.

Return

- ESP_OK : success
- other : failed

Parameters

- [in] `service_handle`: service handle to which this included service is to be added.
- [in] `char_uuid`: : Characteristic UUID.
- [in] `perm`: : Characteristic value declaration attribute permission.
- [in] `property`: : Characteristic Properties
- [in] `char_val`: : Characteristic value
- [in] `control`: : attribute response control byte

`esp_err_t esp_ble_gatts_add_char_descr` (uint16_t *service_handle*, *esp_bt_uuid_t* **descr_uuid*, *esp_gatt_perm_t* *perm*, *esp_attr_value_t* **char_descr_val*, *esp_attr_control_t* **control*)

This function is called to add characteristic descriptor. When it's done, a callback event ESP_GATTS_ADD_DESCR_EVT is called to report the status and an ID number for this descriptor.

Return

- ESP_OK : success
- other : failed

Parameters

- [in] `service_handle`: service handle to which this characteristic descriptor is to be added.
- [in] `perm`: descriptor access permission.
- [in] `descr_uuid`: descriptor UUID.
- [in] `char_descr_val`: : Characteristic descriptor value
- [in] `control`: : attribute response control byte

`esp_err_t esp_ble_gatts_delete_service` (uint16_t *service_handle*)

This function is called to delete a service. When this is done, a callback event ESP_GATTS_DELETE_EVT is report with the status.

Return

- ESP_OK : success
- other : failed

Parameters

- [in] `service_handle`: service_handle to be deleted.

`esp_err_t esp_ble_gatts_start_service` (uint16_t *service_handle*)

This function is called to start a service.

Return

- ESP_OK : success
- other : failed

Parameters

- [in] `service_handle`: the service handle to be started.

`esp_err_t esp_ble_gatts_stop_service` (uint16_t *service_handle*)

This function is called to stop a service.

Return

- `ESP_OK` : success
- other : failed

Parameters

- [in] `service_handle`: - service to be topped.

`esp_err_t esp_ble_gatts_send_indicate` (*esp_gatt_if_t gatts_if*, uint16_t *conn_id*, uint16_t *attr_handle*, uint16_t *value_len*, uint8_t **value*, bool *need_confirm*)

Send indicate or notify to GATT client. Set param `need_confirm` as false will send notification, otherwise indication.

Return

- `ESP_OK` : success
- other : failed

Parameters

- [in] `gatts_if`: GATT server access interface
- [in] `conn_id`: - connection id to indicate.
- [in] `attr_handle`: - attribute handle to indicate.
- [in] `value_len`: - indicate value length.
- [in] `value`: value to indicate.
- [in] `need_confirm`: - Whether a confirmation is required. false sends a GATT notification, true sends a GATT indication.

`esp_err_t esp_ble_gatts_send_response` (*esp_gatt_if_t gatts_if*, uint16_t *conn_id*, uint32_t *trans_id*, *esp_gatt_status_t status*, *esp_gatt_rsp_t *rsp*)

This function is called to send a response to a request.

Return

- `ESP_OK` : success
- other : failed

Parameters

- [in] `gatts_if`: GATT server access interface
- [in] `conn_id`: - connection identifier.
- [in] `trans_id`: - transfer id
- [in] `status`: - response status
- [in] `rsp`: - response data.

`esp_err_t esp_ble_gatts_set_attr_value` (uint16_t *attr_handle*, uint16_t *length*, const uint8_t **value*)

This function is called to set the attribute value by the application.

Return

- `ESP_OK` : success
- other : failed

Parameters

- [in] `attr_handle`: the attribute handle which to be set
- [in] `length`: the value length
- [in] `value`: the pointer to the attribute value

`esp_gatt_status_t esp_ble_gatts_get_attr_value` (uint16_t *attr_handle*, uint16_t **length*, const uint8_t ***value*)

Retrieve attribute value.

Return

- `ESP_GATT_OK` : success
- other : failed

Parameters

- [in] `attr_handle`: Attribute handle.

- [out] length: pointer to the attribute value length
- [out] value: Pointer to attribute value payload, the value cannot be modified by user

esp_err_t **esp_ble_gatts_open** (*esp_gatt_if_t* gatts_if, *esp_bd_addr_t* remote_bda, bool is_direct)

Open a direct open connection or add a background auto connection.

Return

- ESP_OK : success
- other : failed

Parameters

- [in] gatts_if: GATT server access interface
- [in] remote_bda: remote device bluetooth device address.
- [in] is_direct: direct connection or background auto connection

esp_err_t **esp_ble_gatts_close** (*esp_gatt_if_t* gatts_if, uint16_t conn_id)

Close a connection a remote device.

Return

- ESP_OK : success
- other : failed

Parameters

- [in] gatts_if: GATT server access interface
- [in] conn_id: connection ID to be closed.

esp_err_t **esp_ble_gatts_send_service_change_indication** (*esp_gatt_if_t* gatts_if, *esp_bd_addr_t* remote_bda)

Send service change indication.

Return

- ESP_OK : success
- other : failed

Parameters

- [in] gatts_if: GATT server access interface
- [in] remote_bda: remote device bluetooth device address. If remote_bda is NULL then it will send service change indication to all the connected devices and if not then to a specific device

Unions

union esp_ble_gatts_cb_param_t

#include <esp_gatts_api.h> Gatt server callback parameters union.

Public Members

struct esp_ble_gatts_cb_param_t::gatts_reg_evt_param reg

Gatt server callback param of ESP_GATTS_REG_EVT

struct esp_ble_gatts_cb_param_t::gatts_read_evt_param read

Gatt server callback param of ESP_GATTS_READ_EVT

struct esp_ble_gatts_cb_param_t::gatts_write_evt_param write

Gatt server callback param of ESP_GATTS_WRITE_EVT

struct esp_ble_gatts_cb_param_t::gatts_exec_write_evt_param exec_write

Gatt server callback param of ESP_GATTS_EXEC_WRITE_EVT

struct esp_ble_gatts_cb_param_t::gatts_mtu_evt_param mtu

Gatt server callback param of ESP_GATTS_MTU_EVT

struct esp_ble_gatts_cb_param_t::gatts_conf_evt_param conf

Gatt server callback param of ESP_GATTS_CONF_EVT (confirm)

struct esp_ble_gatts_cb_param_t::gatts_create_evt_param create

Gatt server callback param of ESP_GATTS_CREATE_EVT

```

struct esp_ble_gatts_cb_param_t::gatts_add_incl_srvc_evt_param add_incl_srvc
    Gatt server callback param of ESP_GATTS_ADD_INCL_SRVC_EVT

struct esp_ble_gatts_cb_param_t::gatts_add_char_evt_param add_char
    Gatt server callback param of ESP_GATTS_ADD_CHAR_EVT

struct esp_ble_gatts_cb_param_t::gatts_add_char_descr_evt_param add_char_descr
    Gatt server callback param of ESP_GATTS_ADD_CHAR_DESCR_EVT

struct esp_ble_gatts_cb_param_t::gatts_delete_evt_param del
    Gatt server callback param of ESP_GATTS_DELETE_EVT

struct esp_ble_gatts_cb_param_t::gatts_start_evt_param start
    Gatt server callback param of ESP_GATTS_START_EVT

struct esp_ble_gatts_cb_param_t::gatts_stop_evt_param stop
    Gatt server callback param of ESP_GATTS_STOP_EVT

struct esp_ble_gatts_cb_param_t::gatts_connect_evt_param connect
    Gatt server callback param of ESP_GATTS_CONNECT_EVT

struct esp_ble_gatts_cb_param_t::gatts_disconnect_evt_param disconnect
    Gatt server callback param of ESP_GATTS_DISCONNECT_EVT

struct esp_ble_gatts_cb_param_t::gatts_open_evt_param open
    Gatt server callback param of ESP_GATTS_OPEN_EVT

struct esp_ble_gatts_cb_param_t::gatts_cancel_open_evt_param cancel_open
    Gatt server callback param of ESP_GATTS_CANCEL_OPEN_EVT

struct esp_ble_gatts_cb_param_t::gatts_close_evt_param close
    Gatt server callback param of ESP_GATTS_CLOSE_EVT

struct esp_ble_gatts_cb_param_t::gatts_congest_evt_param congest
    Gatt server callback param of ESP_GATTS_CONGEST_EVT

struct esp_ble_gatts_cb_param_t::gatts_rsp_evt_param rsp
    Gatt server callback param of ESP_GATTS_RESPONSE_EVT

struct esp_ble_gatts_cb_param_t::gatts_add_attr_tab_evt_param add_attr_tab
    Gatt server callback param of ESP_GATTS_CREAT_ATTR_TAB_EVT

struct esp_ble_gatts_cb_param_t::gatts_set_attr_val_evt_param set_attr_val
    Gatt server callback param of ESP_GATTS_SET_ATTR_VAL_EVT

struct esp_ble_gatts_cb_param_t::gatts_send_service_change_evt_param service_change
    Gatt server callback param of ESP_GATTS_SEND_SERVICE_CHANGE_EVT

struct gatts_add_attr_tab_evt_param
    #include <esp_gatts_api.h> ESP_GATTS_CREAT_ATTR_TAB_EVT.

```

Public Members

```

esp_gatt_status_t status
    Operation status

esp_bt_uuid_t svc_uuid
    Service uuid type

uint8_t svc_inst_id
    Service id

uint16_t num_handle
    The number of the attribute handle to be added to the gatts database

uint16_t *handles
    The number to the handles

```

```
struct gatts_add_char_descr_evt_param  
#include <esp_gatts_api.h> ESP_GATTS_ADD_CHAR_DESCR_EVT.
```

Public Members

```
esp_gatt_status_t status  
Operation status  
uint16_t attr_handle  
Descriptor attribute handle  
uint16_t service_handle  
Service attribute handle  
esp_bt_uuid_t descr_uuid  
Characteristic descriptor uuid
```

```
struct gatts_add_char_evt_param  
#include <esp_gatts_api.h> ESP_GATTS_ADD_CHAR_EVT.
```

Public Members

```
esp_gatt_status_t status  
Operation status  
uint16_t attr_handle  
Characteristic attribute handle  
uint16_t service_handle  
Service attribute handle  
esp_bt_uuid_t char_uuid  
Characteristic uuid
```

```
struct gatts_add_incl_srvc_evt_param  
#include <esp_gatts_api.h> ESP_GATTS_ADD_INCL_SRVC_EVT.
```

Public Members

```
esp_gatt_status_t status  
Operation status  
uint16_t attr_handle  
Included service attribute handle  
uint16_t service_handle  
Service attribute handle
```

```
struct gatts_cancel_open_evt_param  
#include <esp_gatts_api.h> ESP_GATTS_CANCEL_OPEN_EVT.
```

Public Members

```
esp_gatt_status_t status  
Operation status
```

```
struct gatts_close_evt_param  
#include <esp_gatts_api.h> ESP_GATTS_CLOSE_EVT.
```

Public Members***esp_gatt_status_t* status**

Operation status

uint16_t conn_id

Connection id

struct gatts_conf_evt_param*#include <esp_gatts_api.h>* ESP_GATTS_CONF_EVT.**Public Members*****esp_gatt_status_t* status**

Operation status

uint16_t conn_id

Connection id

uint16_t handle

attribute handle

uint16_t len

The indication or notification value length, len is valid when send notification or indication failed

uint8_t *value

The indication or notification value , value is valid when send notification or indication failed

struct gatts_congest_evt_param*#include <esp_gatts_api.h>* ESP_GATTS_LISTEN_EVT.

ESP_GATTS_CONGEST_EVT

Public Members**uint16_t conn_id**

Connection id

bool congested

Congested or not

struct gatts_connect_evt_param*#include <esp_gatts_api.h>* ESP_GATTS_CONNECT_EVT.**Public Members****uint16_t conn_id**

Connection id

***esp_bd_addr_t* remote_bda**

Remote bluetooth device address

***esp_gatt_conn_params_t* conn_params**

current Connection parameters

struct gatts_create_evt_param*#include <esp_gatts_api.h>* ESP_GATTS_UNREG_EVT.

ESP_GATTS_CREATE_EVT

Public Members

esp_gatt_status_t **status**

Operation status

uint16_t **service_handle**

Service attribute handle

esp_gatt_srv_id_t **service_id**

Service id, include service uuid and other information

struct gatts_delete_evt_param

#include <esp_gatts_api.h> ESP_GATTS_DELETE_EVT.

Public Members

esp_gatt_status_t **status**

Operation status

uint16_t **service_handle**

Service attribute handle

struct gatts_disconnect_evt_param

#include <esp_gatts_api.h> ESP_GATTS_DISCONNECT_EVT.

Public Members

uint16_t **conn_id**

Connection id

esp_bd_addr_t **remote_bda**

Remote bluetooth device address

esp_gatt_conn_reason_t **reason**

Indicate the reason of disconnection

struct gatts_exec_write_evt_param

#include <esp_gatts_api.h> ESP_GATTS_EXEC_WRITE_EVT.

Public Members

uint16_t **conn_id**

Connection id

uint32_t **trans_id**

Transfer id

esp_bd_addr_t **bda**

The bluetooth device address which been written

uint8_t **exec_write_flag**

Execute write flag

struct gatts_mtu_evt_param

#include <esp_gatts_api.h> ESP_GATTS_MTU_EVT.

Public Members

`uint16_t conn_id`
Connection id

`uint16_t mtu`
MTU size

struct gatts_open_evt_param
#include <esp_gatts_api.h> ESP_GATTS_OPEN_EVT.

Public Members

esp_gatt_status_t **status**
Operation status

struct gatts_read_evt_param
#include <esp_gatts_api.h> ESP_GATTS_READ_EVT.

Public Members

`uint16_t conn_id`
Connection id

`uint32_t trans_id`
Transfer id

esp_bd_addr_t **bda**
The bluetooth device address which been read

`uint16_t handle`
The attribute handle

`uint16_t offset`
Offset of the value, if the value is too long

bool **is_long**
The value is too long or not

bool **need_rsp**
The read operation need to do response

struct gatts_reg_evt_param
#include <esp_gatts_api.h> ESP_GATTS_REG_EVT.

Public Members

esp_gatt_status_t **status**
Operation status

`uint16_t app_id`
Application id which input in register API

struct gatts_rsp_evt_param
#include <esp_gatts_api.h> ESP_GATTS_RESPONSE_EVT.

Public Members***esp_gatt_status_t* status**

Operation status

uint16_t handle

Attribute handle which send response

struct gatts_send_service_change_evt_param*#include <esp_gatts_api.h> ESP_GATTS_SEND_SERVICE_CHANGE_EVT.***Public Members*****esp_gatt_status_t* status**

Operation status

struct gatts_set_attr_val_evt_param*#include <esp_gatts_api.h> ESP_GATTS_SET_ATTR_VAL_EVT.***Public Members****uint16_t srvc_handle**

The service handle

uint16_t attr_handle

The attribute handle

***esp_gatt_status_t* status**

Operation status

struct gatts_start_evt_param*#include <esp_gatts_api.h> ESP_GATTS_START_EVT.***Public Members*****esp_gatt_status_t* status**

Operation status

uint16_t service_handle

Service attribute handle

struct gatts_stop_evt_param*#include <esp_gatts_api.h> ESP_GATTS_STOP_EVT.***Public Members*****esp_gatt_status_t* status**

Operation status

uint16_t service_handle

Service attribute handle

struct gatts_write_evt_param*#include <esp_gatts_api.h> ESP_GATTS_WRITE_EVT.*

Public Members

`uint16_t conn_id`
Connection id

`uint32_t trans_id`
Transfer id

`esp_bd_addr_t bda`
The bluetooth device address which been written

`uint16_t handle`
The attribute handle

`uint16_t offset`
Offset of the value, if the value is too long

`bool need_rsp`
The write operation need to do response

`bool is_prep`
This write operation is prepare write

`uint16_t len`
The write attribute value length

`uint8_t *value`
The write attribute value

Macros

`ESP_GATT_PREP_WRITE_CANCEL`
Prepare write flag to indicate cancel prepare write

`ESP_GATT_PREP_WRITE_EXEC`
Prepare write flag to indicate execute prepare write

Type Definitions

```
typedef void (*esp_gatts_cb_t)(esp_gatts_cb_event_t event, esp_gatt_if_t gatts_if,
                              esp_ble_gatts_cb_param_t *param)
```

GATT Server callback function type.

Parameters

- `event`: : Event type
- `gatts_if`: : GATT server access interface, normally different `gatts_if` correspond to different profile
- `param`: : Point to callback parameter, currently is union type

Enumerations

```
enum esp_gatts_cb_event_t
```

GATT Server callback function events.

Values:

`ESP_GATTS_REG_EVT = 0`
When register application id, the event comes

`ESP_GATTS_READ_EVT = 1`
When gatt client request read operation, the event comes

`ESP_GATTS_WRITE_EVT = 2`
When gatt client request write operation, the event comes

`ESP_GATTS_EXEC_WRITE_EVT = 3`
When gatt client request execute write, the event comes

- ESP_GATTS_MTU_EVT = 4**
When set mtu complete, the event comes
- ESP_GATTS_CONF_EVT = 5**
When receive confirm, the event comes
- ESP_GATTS_UNREG_EVT = 6**
When unregister application id, the event comes
- ESP_GATTS_CREATE_EVT = 7**
When create service complete, the event comes
- ESP_GATTS_ADD_INCL_SRVC_EVT = 8**
When add included service complete, the event comes
- ESP_GATTS_ADD_CHAR_EVT = 9**
When add characteristic complete, the event comes
- ESP_GATTS_ADD_CHAR_DESCR_EVT = 10**
When add descriptor complete, the event comes
- ESP_GATTS_DELETE_EVT = 11**
When delete service complete, the event comes
- ESP_GATTS_START_EVT = 12**
When start service complete, the event comes
- ESP_GATTS_STOP_EVT = 13**
When stop service complete, the event comes
- ESP_GATTS_CONNECT_EVT = 14**
When gatt client connect, the event comes
- ESP_GATTS_DISCONNECT_EVT = 15**
When gatt client disconnect, the event comes
- ESP_GATTS_OPEN_EVT = 16**
When connect to peer, the event comes
- ESP_GATTS_CANCEL_OPEN_EVT = 17**
When disconnect from peer, the event comes
- ESP_GATTS_CLOSE_EVT = 18**
When gatt server close, the event comes
- ESP_GATTS_LISTEN_EVT = 19**
When gatt listen to be connected the event comes
- ESP_GATTS_CONGEST_EVT = 20**
When congest happen, the event comes
- ESP_GATTS_RESPONSE_EVT = 21**
When gatt send response complete, the event comes
- ESP_GATTS_CREAT_ATTR_TAB_EVT = 22**
When gatt create table complete, the event comes
- ESP_GATTS_SET_ATTR_VAL_EVT = 23**
When gatt set attr value complete, the event comes
- ESP_GATTS_SEND_SERVICE_CHANGE_EVT = 24**
When gatt send service change indication complete, the event comes

GATT CLIENT API

[Overview](#) [Instructions](#)

Application Example Check [bluetooth/bluedroid/ble](#) folder in ESP-IDF examples, which contains the following demos and their tutorials:

- This is a GATT client demo and its tutorial. This demo can scan for devices, connect to the GATT server and discover its services.
 - [bluetooth/bluedroid/ble/gatt_client](#)
 - [GATT Client Example Walkthrough](#)
- This is a multiple connection demo and its tutorial. This demo can connect to multiple GATT server devices and discover their services.
 - [bluetooth/bluedroid/ble/gattc_multi_connect](#)
 - [GATT Client Multi-connection Example Walkthrough](#)
- This is a BLE SPP-Like demo. This demo, which acts as a GATT client, can receive data from UART and then send the data to the peer device automatically.
 - [bluetooth/bluedroid/ble/ble_spp_client](#)

API Reference

Header File

- [bt/host/bluedroid/api/include/api/esp_gattc_api.h](#)

Functions

esp_err_t **esp_ble_gattc_register_callback** (*esp_gattc_cb_t* callback)

This function is called to register application callbacks with GATTC module.

Return

- ESP_OK: success
- other: failed

Parameters

- [in] callback: : pointer to the application callback function.

esp_err_t **esp_ble_gattc_app_register** (uint16_t app_id)

This function is called to register application callbacks with GATTC module.

Return

- ESP_OK: success
- other: failed

Parameters

- [in] app_id: : Application Identify (UUID), for different application

esp_err_t **esp_ble_gattc_app_unregister** (*esp_gatt_if_t* gattc_if)

This function is called to unregister an application from GATTC module.

Return

- ESP_OK: success
- other: failed

Parameters

- [in] gattc_if: Gatt client access interface.

esp_err_t **esp_ble_gattc_open** (*esp_gatt_if_t* gattc_if, *esp_bd_addr_t* remote_bda, *esp_ble_addr_type_t* remote_addr_type, bool is_direct)

Open a direct connection or add a background auto connection.

Return

- ESP_OK: success
- other: failed

Parameters

- [in] gattc_if: Gatt client access interface.
- [in] remote_bda: remote device bluetooth device address.
- [in] remote_addr_type: remote device bluetooth device the address type.

- [in] `is_direct`: direct connection or background auto connection (by now, background auto connection is not supported).

esp_err_t **esp_ble_gattc_close** (*esp_gatt_if_t* *gattc_if*, *uint16_t* *conn_id*)

Close the virtual connection to the GATT server. `gattc` may have multiple virtual GATT server connections when multiple `app_id` registered, this API only close one virtual GATT server connection. if there exist other virtual GATT server connections, it does not disconnect the physical connection. if you want to disconnect the physical connection directly, you can use `esp_ble_gap_disconnect(esp_bd_addr_t remote_device)`.

Return

- `ESP_OK`: success
- other: failed

Parameters

- [in] `gattc_if`: Gatt client access interface.
- [in] `conn_id`: connection ID to be closed.

esp_err_t **esp_ble_gattc_send_mtu_req** (*esp_gatt_if_t* *gattc_if*, *uint16_t* *conn_id*)

Configure the MTU size in the GATT channel. This can be done only once per connection. Before using, use `esp_ble_gatt_set_local_mtu()` to configure the local MTU size.

Return

- `ESP_OK`: success
- other: failed

Parameters

- [in] `gattc_if`: Gatt client access interface.
- [in] `conn_id`: connection ID.

esp_err_t **esp_ble_gattc_search_service** (*esp_gatt_if_t* *gattc_if*, *uint16_t* *conn_id*, *esp_bt_uuid_t* **filter_uuid*)

This function is called to get service from local cache. This function report service search result by a callback event, and followed by a service search complete event.

Return

- `ESP_OK`: success
- other: failed

Parameters

- [in] `gattc_if`: Gatt client access interface.
- [in] `conn_id`: connection ID.
- [in] `filter_uuid`: a UUID of the service application is interested in. If Null, discover for all services.

esp_gatt_status_t **esp_ble_gattc_get_service** (*esp_gatt_if_t* *gattc_if*, *uint16_t* *conn_id*, *esp_bt_uuid_t* **svc_uuid*, *esp_gattc_service_elem_t* **result*, *uint16_t* **count*, *uint16_t* *offset*)

Find all the service with the given service uuid in the `gattc` cache, if the `svc_uuid` is NULL, find all the service. Note: It just get service from local cache, won't get from remote devices. If want to get it from remote device, need to used the `esp_ble_gattc_cache_refresh`, then call `esp_ble_gattc_get_service` again.

Return

- `ESP_OK`: success
- other: failed

Parameters

- [in] `gattc_if`: Gatt client access interface.
- [in] `conn_id`: connection ID which identify the server.
- [in] `svc_uuid`: the pointer to the service uuid.
- [out] `result`: The pointer to the service which has been found in the `gattc` cache.
- [inout] `count`: input the number of service want to find, it will output the number of service has been found in the `gattc` cache with the given service uuid.
- [in] `offset`: Offset of the service position to get.

```
esp_gatt_status_t esp_ble_gattc_get_all_char(esp_gatt_if_t gattc_if, uint16_t conn_id,
uint16_t start_handle, uint16_t end_handle,
esp_gattc_char_elem_t *result, uint16_t *count,
uint16_t offset)
```

Find all the characteristic with the given service in the gattc cache Note: It just get characteristic from local cache, won't get from remote devices.

Return

- ESP_OK: success
- other: failed

Parameters

- [in] gattc_if: Gatt client access interface.
- [in] conn_id: connection ID which identify the server.
- [in] start_handle: the attribute start handle.
- [in] end_handle: the attribute end handle
- [out] result: The pointer to the characteristic in the service.
- [inout] count: input the number of characteristic want to find, it will output the number of characteristic has been found in the gattc cache with the given service.
- [in] offset: Offset of the characteristic position to get.

```
esp_gatt_status_t esp_ble_gattc_get_all_descr(esp_gatt_if_t gattc_if, uint16_t conn_id, uint16_t
char_handle, esp_gattc_descr_elem_t *result,
uint16_t *count, uint16_t offset)
```

Find all the descriptor with the given characteristic in the gattc cache Note: It just get descriptor from local cache, won't get from remote devices.

Return

- ESP_OK: success
- other: failed

Parameters

- [in] gattc_if: Gatt client access interface.
- [in] conn_id: connection ID which identify the server.
- [in] char_handle: the given characteristic handle
- [out] result: The pointer to the descriptor in the characteristic.
- [inout] count: input the number of descriptor want to find, it will output the number of descriptor has been found in the gattc cache with the given characteristic.
- [in] offset: Offset of the descriptor position to get.

```
esp_gatt_status_t esp_ble_gattc_get_char_by_uuid(esp_gatt_if_t gattc_if, uint16_t conn_id,
uint16_t start_handle, uint16_t
end_handle, esp_bt_uuid_t char_uuid,
esp_gattc_char_elem_t *result, uint16_t
*count)
```

Find the characteristic with the given characteristic uuid in the gattc cache Note: It just get characteristic from local cache, won't get from remote devices.

Return

- ESP_OK: success
- other: failed

Parameters

- [in] gattc_if: Gatt client access interface.
- [in] conn_id: connection ID which identify the server.
- [in] start_handle: the attribute start handle
- [in] end_handle: the attribute end handle
- [in] char_uuid: the characteristic uuid
- [out] result: The pointer to the characteristic in the service.
- [inout] count: input the number of characteristic want to find, it will output the number of characteristic has been found in the gattc cache with the given service.

```
esp_gatt_status_t esp_ble_gattc_get_descr_by_uuid (esp_gatt_if_t gattc_if, uint16_t conn_id, uint16_t start_handle, uint16_t end_handle, esp_bt_uuid_t char_uuid, esp_bt_uuid_t descr_uuid, esp_gattc_descr_elem_t *result, uint16_t *count)
```

Find the descriptor with the given characteristic uuid in the gattc cache Note: It just get descriptor from local cache, won't get from remote devices.

Return

- ESP_OK: success
- other: failed

Parameters

- [in] gattc_if: Gatt client access interface.
- [in] conn_id: connection ID which identify the server.
- [in] start_handle: the attribute start handle
- [in] end_handle: the attribute end handle
- [in] char_uuid: the characteristic uuid.
- [in] descr_uuid: the descriptor uuid.
- [out] result: The pointer to the descriptor in the given characteristic.
- [inout] count: input the number of descriptor want to find, it will output the number of descriptor has been found in the gattc cache with the given characteristic.

```
esp_gatt_status_t esp_ble_gattc_get_descr_by_char_handle (esp_gatt_if_t gattc_if, uint16_t conn_id, uint16_t char_handle, esp_bt_uuid_t descr_uuid, esp_gattc_descr_elem_t *result, uint16_t *count)
```

Find the descriptor with the given characteristic handle in the gattc cache Note: It just get descriptor from local cache, won't get from remote devices.

Return

- ESP_OK: success
- other: failed

Parameters

- [in] gattc_if: Gatt client access interface.
- [in] conn_id: connection ID which identify the server.
- [in] char_handle: the characteristic handle.
- [in] descr_uuid: the descriptor uuid.
- [out] result: The pointer to the descriptor in the given characteristic.
- [inout] count: input the number of descriptor want to find, it will output the number of descriptor has been found in the gattc cache with the given characteristic.

```
esp_gatt_status_t esp_ble_gattc_get_include_service (esp_gatt_if_t gattc_if, uint16_t conn_id, uint16_t start_handle, uint16_t end_handle, esp_bt_uuid_t *incl_uuid, esp_gattc_incl_svc_elem_t *result, uint16_t *count)
```

Find the include service with the given service handle in the gattc cache Note: It just get include service from local cache, won't get from remote devices.

Return

- ESP_OK: success
- other: failed

Parameters

- [in] gattc_if: Gatt client access interface.
- [in] conn_id: connection ID which identify the server.
- [in] start_handle: the attribute start handle
- [in] end_handle: the attribute end handle
- [in] incl_uuid: the include service uuid
- [out] result: The pointer to the include service in the given service.

- [inout] `count`: input the number of include service want to find, it will output the number of include service has been found in the gattc cache with the given service.

```
esp_gatt_status_t esp_ble_gattc_get_attr_count(esp_gatt_if_t gattc_if, uint16_t conn_id,
                                             esp_gatt_db_attr_type_t type, uint16_t
                                             start_handle, uint16_t end_handle, uint16_t
                                             char_handle, uint16_t *count)
```

Find the attribute count with the given service or characteristic in the gattc cache.

Return

- ESP_OK: success
- other: failed

Parameters

- [in] `gattc_if`: Gatt client access interface.
- [in] `conn_id`: connection ID which identify the server.
- [in] `type`: the attribute type.
- [in] `start_handle`: the attribute start handle, if the type is ESP_GATT_DB_DESCRIPTOR, this parameter should be ignore
- [in] `end_handle`: the attribute end handle, if the type is ESP_GATT_DB_DESCRIPTOR, this parameter should be ignore
- [in] `char_handle`: the characteristic handle, this parameter valid when the type is ESP_GATT_DB_DESCRIPTOR. If the type isn't ESP_GATT_DB_DESCRIPTOR, this parameter should be ignore.
- [out] `count`: output the number of attribute has been found in the gattc cache with the given attribute type.

```
esp_gatt_status_t esp_ble_gattc_get_db(esp_gatt_if_t gattc_if, uint16_t conn_id, uint16_t
                                       start_handle, uint16_t end_handle, esp_gattc_db_elem_t
                                       *db, uint16_t *count)
```

This function is called to get the GATT database. Note: It just get attribute data base from local cache, won't get from remote devices.

Return

- ESP_OK: success
- other: failed

Parameters

- [in] `gattc_if`: Gatt client access interface.
- [in] `start_handle`: the attribute start handle
- [in] `end_handle`: the attribute end handle
- [in] `conn_id`: connection ID which identify the server.
- [in] `db`: output parameter which will contain the GATT database copy. Caller is responsible for freeing it.
- [in] `count`: number of elements in database.

```
esp_err_t esp_ble_gattc_read_char(esp_gatt_if_t gattc_if, uint16_t conn_id, uint16_t handle,
                                  esp_gatt_auth_req_t auth_req)
```

This function is called to read a service's characteristics of the given characteristic handle.

Return

- ESP_OK: success
- other: failed

Parameters

- [in] `gattc_if`: Gatt client access interface.
- [in] `conn_id`: connection ID.
- [in] `handle`: characteristic handle to read.
- [in] `auth_req`: authenticate request type

```
esp_err_t esp_ble_gattc_read_by_type(esp_gatt_if_t gattc_if, uint16_t conn_id, uint16_t
                                       start_handle, uint16_t end_handle, esp_bt_uuid_t *uuid,
                                       esp_gatt_auth_req_t auth_req)
```

This function is called to read a service's characteristics of the given characteristic UUID.

Return

- ESP_OK: success
- other: failed

Parameters

- [in] *gattc_if*: Gatt client access interface.
- [in] *conn_id*: : connection ID.
- [in] *start_handle*: : the attribute start handle.
- [in] *end_handle*: : the attribute end handle
- [in] *uuid*: : The UUID of attribute which will be read.
- [in] *auth_req*: : authenticate request type

```
esp_err_t esp_ble_gattc_read_multiple(esp_gatt_if_t gattc_if, uint16_t conn_id,
                                     esp_gattc_multi_t *read_multi, esp_gatt_auth_req_t
                                     auth_req)
```

This function is called to read multiple characteristic or characteristic descriptors.

Return

- ESP_OK: success
- other: failed

Parameters

- [in] *gattc_if*: Gatt client access interface.
- [in] *conn_id*: : connection ID.
- [in] *read_multi*: : pointer to the read multiple parameter.
- [in] *auth_req*: : authenticate request type

```
esp_err_t esp_ble_gattc_read_char_descr(esp_gatt_if_t gattc_if, uint16_t conn_id, uint16_t handle,
                                         esp_gatt_auth_req_t auth_req)
```

This function is called to read a characteristics descriptor.

Return

- ESP_OK: success
- other: failed

Parameters

- [in] *gattc_if*: Gatt client access interface.
- [in] *conn_id*: : connection ID.
- [in] *handle*: : descriptor handle to read.
- [in] *auth_req*: : authenticate request type

```
esp_err_t esp_ble_gattc_write_char(esp_gatt_if_t gattc_if, uint16_t conn_id, uint16_t handle,
                                   uint16_t value_len, uint8_t *value, esp_gatt_write_type_t
                                   write_type, esp_gatt_auth_req_t auth_req)
```

This function is called to write characteristic value.

Return

- ESP_OK: success
- other: failed

Parameters

- [in] *gattc_if*: Gatt client access interface.
- [in] *conn_id*: : connection ID.
- [in] *handle*: : characteristic handle to write.
- [in] *value_len*: length of the value to be written.
- [in] *value*: : the value to be written.
- [in] *write_type*: : the type of attribute write operation.
- [in] *auth_req*: : authentication request.

```
esp_err_t esp_ble_gattc_write_char_descr(esp_gatt_if_t gattc_if, uint16_t conn_id,
                                          uint16_t handle, uint16_t value_len, uint8_t
                                          *value, esp_gatt_write_type_t write_type,
                                          esp_gatt_auth_req_t auth_req)
```

This function is called to write characteristic descriptor value.

Return

- ESP_OK: success
- other: failed

Parameters

- [in] `gattc_if`: Gatt client access interface.
- [in] `conn_id`: : connection ID
- [in] `handle`: : descriptor handle to write.
- [in] `value_len`: length of the value to be written.
- [in] `value`: : the value to be written.
- [in] `write_type`: : the type of attribute write operation.
- [in] `auth_req`: : authentication request.

`esp_err_t esp_ble_gattc_prepare_write(esp_gatt_if_t gattc_if, uint16_t conn_id, uint16_t handle, uint16_t offset, uint16_t value_len, uint8_t *value, esp_gatt_auth_req_t auth_req)`

This function is called to prepare write a characteristic value.

Return

- `ESP_OK`: success
- other: failed

Parameters

- [in] `gattc_if`: Gatt client access interface.
- [in] `conn_id`: : connection ID.
- [in] `handle`: : characteristic handle to prepare write.
- [in] `offset`: : offset of the write value.
- [in] `value_len`: length of the value to be written.
- [in] `value`: : the value to be written.
- [in] `auth_req`: : authentication request.

`esp_err_t esp_ble_gattc_prepare_write_char_descr(esp_gatt_if_t gattc_if, uint16_t conn_id, uint16_t handle, uint16_t offset, uint16_t value_len, uint8_t *value, esp_gatt_auth_req_t auth_req)`

This function is called to prepare write a characteristic descriptor value.

Return

- `ESP_OK`: success
- other: failed

Parameters

- [in] `gattc_if`: Gatt client access interface.
- [in] `conn_id`: : connection ID.
- [in] `handle`: : characteristic descriptor handle to prepare write.
- [in] `offset`: : offset of the write value.
- [in] `value_len`: length of the value to be written.
- [in] `value`: : the value to be written.
- [in] `auth_req`: : authentication request.

`esp_err_t esp_ble_gattc_execute_write(esp_gatt_if_t gattc_if, uint16_t conn_id, bool is_execute)`

This function is called to execute write a prepare write sequence.

Return

- `ESP_OK`: success
- other: failed

Parameters

- [in] `gattc_if`: Gatt client access interface.
- [in] `conn_id`: : connection ID.
- [in] `is_execute`: : execute or cancel.

`esp_err_t esp_ble_gattc_register_for_notify(esp_gatt_if_t gattc_if, esp_bd_addr_t server_bda, uint16_t handle)`

This function is called to register for notification of a service.

Return

- `ESP_OK`: registration succeeds
- other: failed

Parameters

- [in] `gattc_if`: Gatt client access interface.
- [in] `server_bda`: : target GATT server.
- [in] `handle`: : GATT characteristic handle.

esp_err_t **esp_ble_gattc_unregister_for_notify** (*esp_gatt_if_t* `gattc_if`, *esp_bd_addr_t* `server_bda`, `uint16_t handle`)

This function is called to de-register for notification of a service.

Return

- ESP_OK: unregister succeeds
- other: failed

Parameters

- [in] `gattc_if`: Gatt client access interface.
- [in] `server_bda`: : target GATT server.
- [in] `handle`: : GATT characteristic handle.

esp_err_t **esp_ble_gattc_cache_refresh** (*esp_bd_addr_t* `remote_bda`)

Refresh the server cache store in the gattc stack of the remote device. If the device is connected, this API will restart the discovery of service information of the remote device.

Return

- ESP_OK: success
- other: failed

Parameters

- [in] `remote_bda`: remote device BD address.

esp_err_t **esp_ble_gattc_cache_assoc** (*esp_gatt_if_t* `gattc_if`, *esp_bd_addr_t* `src_addr`, *esp_bd_addr_t* `assoc_addr`, `bool is_assoc`)

Add or delete the associated address with the source address. Note: The role of this API is mainly when the client side has stored a server-side database, when it needs to connect another device, but the device's attribute database is the same as the server database stored on the client-side, calling this API can use the database that the device has stored used as the peer server database to reduce the attribute database search and discovery process and speed up the connection time. The associated address means that device want to use the database has stored in the local cache. The source address means that device want to share the database to the associated address device.

Return

- ESP_OK: success
- other: failed

Parameters

- [in] `gattc_if`: Gatt client access interface.
- [in] `src_addr`: the source address which provide the attribute table.
- [in] `assoc_addr`: the associated device address which went to share the attribute table with the source address.
- [in] `is_assoc`: true add the associated device address, false remove the associated device address.

esp_err_t **esp_ble_gattc_cache_get_addr_list** (*esp_gatt_if_t* `gattc_if`)

Get the address list which has store the attribute table in the gattc cache. There will callback ESP_GATTC_GET_ADDR_LIST_EVT event when get address list complete.

Return

- ESP_OK: success
- other: failed

Parameters

- [in] `gattc_if`: Gatt client access interface.

esp_err_t **esp_ble_gattc_cache_clean** (*esp_bd_addr_t* `remote_bda`)

Clean the service cache of this device in the gattc stack,.

Return

- ESP_OK: success
- other: failed

Parameters

- [in] remote_bda: remote device BD address.

Unions

union esp_ble_gattc_cb_param_t

#include <esp_gattc_api.h> Gatt client callback parameters union.

Public Members

struct esp_ble_gattc_cb_param_t::gattc_reg_evt_param reg
Gatt client callback param of ESP_GATTC_REG_EVT

struct esp_ble_gattc_cb_param_t::gattc_open_evt_param open
Gatt client callback param of ESP_GATTC_OPEN_EVT

struct esp_ble_gattc_cb_param_t::gattc_close_evt_param close
Gatt client callback param of ESP_GATTC_CLOSE_EVT

struct esp_ble_gattc_cb_param_t::gattc_cfg_mtu_evt_param cfg_mtu
Gatt client callback param of ESP_GATTC_CFG_MTU_EVT

struct esp_ble_gattc_cb_param_t::gattc_search_cmpl_evt_param search_cmpl
Gatt client callback param of ESP_GATTC_SEARCH_CMPL_EVT

struct esp_ble_gattc_cb_param_t::gattc_search_res_evt_param search_res
Gatt client callback param of ESP_GATTC_SEARCH_RES_EVT

struct esp_ble_gattc_cb_param_t::gattc_read_char_evt_param read
Gatt client callback param of ESP_GATTC_READ_CHAR_EVT

struct esp_ble_gattc_cb_param_t::gattc_write_evt_param write
Gatt client callback param of ESP_GATTC_WRITE_DESCR_EVT

struct esp_ble_gattc_cb_param_t::gattc_exec_cmpl_evt_param exec_cmpl
Gatt client callback param of ESP_GATTC_EXEC_EVT

struct esp_ble_gattc_cb_param_t::gattc_notify_evt_param notify
Gatt client callback param of ESP_GATTC_NOTIFY_EVT

struct esp_ble_gattc_cb_param_t::gattc_srv_chg_evt_param srv_chg
Gatt client callback param of ESP_GATTC_SRVC_CHG_EVT

struct esp_ble_gattc_cb_param_t::gattc_congest_evt_param congest
Gatt client callback param of ESP_GATTC_CONGEST_EVT

struct esp_ble_gattc_cb_param_t::gattc_reg_for_notify_evt_param reg_for_notify
Gatt client callback param of ESP_GATTC_REG_FOR_NOTIFY_EVT

struct esp_ble_gattc_cb_param_t::gattc_unreg_for_notify_evt_param unreg_for_notify
Gatt client callback param of ESP_GATTC_UNREG_FOR_NOTIFY_EVT

struct esp_ble_gattc_cb_param_t::gattc_connect_evt_param connect
Gatt client callback param of ESP_GATTC_CONNECT_EVT

struct esp_ble_gattc_cb_param_t::gattc_disconnect_evt_param disconnect
Gatt client callback param of ESP_GATTC_DISCONNECT_EVT

struct esp_ble_gattc_cb_param_t::gattc_set_assoc_addr_cmp_evt_param set_assoc_cmp
Gatt client callback param of ESP_GATTC_SET_ASSOC_EVT

struct esp_ble_gattc_cb_param_t::gattc_get_addr_list_evt_param get_addr_list
Gatt client callback param of ESP_GATTC_GET_ADDR_LIST_EVT

struct esp_ble_gattc_cb_param_t::gattc_queue_full_evt_param queue_full
Gatt client callback param of ESP_GATTC_QUEUE_FULL_EVT

```
struct esp_ble_gattc_cb_param_t::gattc_dis_srvc_cmpl_evt_param dis_srvc_cmpl  
    Gatt client callback param of ESP_GATTC_DIS_SRVC_CMPL_EVT
```

```
struct gattc_cfg_mtu_evt_param  
    #include <esp_gattc_api.h> ESP_GATTC_CFG_MTU_EVT.
```

Public Members

```
esp_gatt_status_t status  
    Operation status
```

```
uint16_t conn_id  
    Connection id
```

```
uint16_t mtu  
    MTU size
```

```
struct gattc_close_evt_param  
    #include <esp_gattc_api.h> ESP_GATTC_CLOSE_EVT.
```

Public Members

```
esp_gatt_status_t status  
    Operation status
```

```
uint16_t conn_id  
    Connection id
```

```
esp_bd_addr_t remote_bda  
    Remote bluetooth device address
```

```
esp_gatt_conn_reason_t reason  
    The reason of gatt connection close
```

```
struct gattc_congest_evt_param  
    #include <esp_gattc_api.h> ESP_GATTC_CONGEST_EVT.
```

Public Members

```
uint16_t conn_id  
    Connection id
```

```
bool congested  
    Congested or not
```

```
struct gattc_connect_evt_param  
    #include <esp_gattc_api.h> ESP_GATTC_CONNECT_EVT.
```

Public Members

```
uint16_t conn_id  
    Connection id
```

```
esp_bd_addr_t remote_bda  
    Remote bluetooth device address
```

```
esp_gatt_conn_params_t conn_params  
    current connection parameters
```

```
struct gattc_dis_srvc_cmpl_evt_param  
    #include <esp_gattc_api.h> ESP_GATTC_DIS_SRVC_CMPL_EVT.
```

Public Members

esp_gatt_status_t **status**

Operation status

uint16_t **conn_id**

Connection id

struct gattc_disconnect_evt_param

#include <esp_gattc_api.h> ESP_GATTC_DISCONNECT_EVT.

Public Members

esp_gatt_conn_reason_t **reason**

disconnection reason

uint16_t **conn_id**

Connection id

esp_bd_addr_t **remote_bda**

Remote bluetooth device address

struct gattc_exec_cmpl_evt_param

#include <esp_gattc_api.h> ESP_GATTC_EXEC_EVT.

Public Members

esp_gatt_status_t **status**

Operation status

uint16_t **conn_id**

Connection id

struct gattc_get_addr_list_evt_param

#include <esp_gattc_api.h> ESP_GATTC_GET_ADDR_LIST_EVT.

Public Members

esp_gatt_status_t **status**

Operation status

uint8_t **num_addr**

The number of address in the gattc cache address list

esp_bd_addr_t ***addr_list**

The pointer to the address list which has been get from the gattc cache

struct gattc_notify_evt_param

#include <esp_gattc_api.h> ESP_GATTC_NOTIFY_EVT.

Public Members

uint16_t **conn_id**

Connection id

esp_bd_addr_t **remote_bda**

Remote bluetooth device address

uint16_t **handle**

The Characteristic or descriptor handle

`uint16_t value_len`
Notify attribute value

`uint8_t *value`
Notify attribute value

`bool is_notify`
True means notify, false means indicate

struct gattc_open_evt_param
#include <esp_gattc_api.h> ESP_GATTC_OPEN_EVT.

Public Members

esp_gatt_status_t **status**
Operation status

`uint16_t conn_id`
Connection id

esp_bd_addr_t **remote_bda**
Remote bluetooth device address

`uint16_t mtu`
MTU size

struct gattc_queue_full_evt_param
#include <esp_gattc_api.h> ESP_GATTC_QUEUE_FULL_EVT.

Public Members

esp_gatt_status_t **status**
Operation status

`uint16_t conn_id`
Connection id

`bool is_full`
The gattc command queue is full or not

struct gattc_read_char_evt_param
#include <esp_gattc_api.h> ESP_GATTC_READ_CHAR_EVT, ESP_GATTC_READ_DESCR_EVT.

Public Members

esp_gatt_status_t **status**
Operation status

`uint16_t conn_id`
Connection id

`uint16_t handle`
Characteristic handle

`uint8_t *value`
Characteristic value

`uint16_t value_len`
Characteristic value length

struct gattc_reg_evt_param
#include <esp_gattc_api.h> ESP_GATTC_REG_EVT.

Public Members

esp_gatt_status_t **status**

Operation status

uint16_t **app_id**

Application id which input in register API

struct gattc_reg_for_notify_evt_param

#include <esp_gattc_api.h> ESP_GATTC_REG_FOR_NOTIFY_EVT.

Public Members

esp_gatt_status_t **status**

Operation status

uint16_t **handle**

The characteristic or descriptor handle

struct gattc_search_cmpl_evt_param

#include <esp_gattc_api.h> ESP_GATTC_SEARCH_CMPL_EVT.

Public Members

esp_gatt_status_t **status**

Operation status

uint16_t **conn_id**

Connection id

esp_service_source_t **searched_service_source**

The source of the service information

struct gattc_search_res_evt_param

#include <esp_gattc_api.h> ESP_GATTC_SEARCH_RES_EVT.

Public Members

uint16_t **conn_id**

Connection id

uint16_t **start_handle**

Service start handle

uint16_t **end_handle**

Service end handle

esp_gatt_id_t **srvc_id**

Service id, include service uuid and other information

bool **is_primary**

True if this is the primary service

struct gattc_set_assoc_addr_cmp_evt_param

#include <esp_gattc_api.h> ESP_GATTC_SET_ASSOC_EVT.

Public Members

esp_gatt_status_t **status**
Operation status

struct gattc_srvc_chg_evt_param
#include <esp_gattc_api.h> ESP_GATTC_SRVC_CHG_EVT.

Public Members

esp_bd_addr_t **remote_bda**
Remote bluetooth device address

struct gattc_unreg_for_notify_evt_param
#include <esp_gattc_api.h> ESP_GATTC_UNREG_FOR_NOTIFY_EVT.

Public Members

esp_gatt_status_t **status**
Operation status

uint16_t **handle**
The characteristic or descriptor handle

struct gattc_write_evt_param
#include <esp_gattc_api.h> ESP_GATTC_WRITE_CHAR_EVT, ESP_GATTC_PREP_WRITE_EVT,
ESP_GATTC_WRITE_DESCR_EVT.

Public Members

esp_gatt_status_t **status**
Operation status

uint16_t **conn_id**
Connection id

uint16_t **handle**
The Characteristic or descriptor handle

uint16_t **offset**
The prepare write offset, this value is valid only when prepare write

Type Definitions

```
typedef void (*esp_gattc_cb_t) (esp_gattc_cb_event_t event, esp_gatt_if_t gattc_if,  
esp_ble_gattc_cb_param_t *param)
```

GATT Client callback function type.

Parameters

- *event*: : Event type
- *gattc_if*: : GATT client access interface, normally different *gattc_if* correspond to different profile
- *param*: : Point to callback parameter, currently is union type

Enumerations

enum esp_gattc_cb_event_t
GATT Client callback function events.

Values:

- ESP_GATTC_REG_EVT = 0**
When GATT client is registered, the event comes
- ESP_GATTC_UNREG_EVT = 1**
When GATT client is unregistered, the event comes
- ESP_GATTC_OPEN_EVT = 2**
When GATT virtual connection is set up, the event comes
- ESP_GATTC_READ_CHAR_EVT = 3**
When GATT characteristic is read, the event comes
- ESP_GATTC_WRITE_CHAR_EVT = 4**
When GATT characteristic write operation completes, the event comes
- ESP_GATTC_CLOSE_EVT = 5**
When GATT virtual connection is closed, the event comes
- ESP_GATTC_SEARCH_CMPL_EVT = 6**
When GATT service discovery is completed, the event comes
- ESP_GATTC_SEARCH_RES_EVT = 7**
When GATT service discovery result is got, the event comes
- ESP_GATTC_READ_DESCR_EVT = 8**
When GATT characteristic descriptor read completes, the event comes
- ESP_GATTC_WRITE_DESCR_EVT = 9**
When GATT characteristic descriptor write completes, the event comes
- ESP_GATTC_NOTIFY_EVT = 10**
When GATT notification or indication arrives, the event comes
- ESP_GATTC_PREP_WRITE_EVT = 11**
When GATT prepare-write operation completes, the event comes
- ESP_GATTC_EXEC_EVT = 12**
When write execution completes, the event comes
- ESP_GATTC_ACL_EVT = 13**
When ACL connection is up, the event comes
- ESP_GATTC_CANCEL_OPEN_EVT = 14**
When GATT client ongoing connection is cancelled, the event comes
- ESP_GATTC_SRVC_CHG_EVT = 15**
When “service changed” occurs, the event comes
- ESP_GATTC_ENC_CMPL_CB_EVT = 17**
When encryption procedure completes, the event comes
- ESP_GATTC_CFG_MTU_EVT = 18**
When configuration of MTU completes, the event comes
- ESP_GATTC_ADV_DATA_EVT = 19**
When advertising of data, the event comes
- ESP_GATTC_MULT_ADV_ENB_EVT = 20**
When multi-advertising is enabled, the event comes
- ESP_GATTC_MULT_ADV_UPD_EVT = 21**
When multi-advertising parameters are updated, the event comes
- ESP_GATTC_MULT_ADV_DATA_EVT = 22**
When multi-advertising data arrives, the event comes
- ESP_GATTC_MULT_ADV_DIS_EVT = 23**
When multi-advertising is disabled, the event comes

ESP_GATTC_CONGEST_EVT = 24

When GATT connection congestion comes, the event comes

ESP_GATTC_BTH_SCAN_ENB_EVT = 25

When batch scan is enabled, the event comes

ESP_GATTC_BTH_SCAN_CFG_EVT = 26

When batch scan storage is configured, the event comes

ESP_GATTC_BTH_SCAN_RD_EVT = 27

When Batch scan read event is reported, the event comes

ESP_GATTC_BTH_SCAN_THR_EVT = 28

When Batch scan threshold is set, the event comes

ESP_GATTC_BTH_SCAN_PARAM_EVT = 29

When Batch scan parameters are set, the event comes

ESP_GATTC_BTH_SCAN_DIS_EVT = 30

When Batch scan is disabled, the event comes

ESP_GATTC_SCAN_FLT_CFG_EVT = 31

When Scan filter configuration completes, the event comes

ESP_GATTC_SCAN_FLT_PARAM_EVT = 32

When Scan filter parameters are set, the event comes

ESP_GATTC_SCAN_FLT_STATUS_EVT = 33

When Scan filter status is reported, the event comes

ESP_GATTC_ADV_VSC_EVT = 34

When advertising vendor spec content event is reported, the event comes

ESP_GATTC_REG_FOR_NOTIFY_EVT = 38

When register for notification of a service completes, the event comes

ESP_GATTC_UNREG_FOR_NOTIFY_EVT = 39

When unregister for notification of a service completes, the event comes

ESP_GATTC_CONNECT_EVT = 40

When the ble physical connection is set up, the event comes

ESP_GATTC_DISCONNECT_EVT = 41

When the ble physical connection disconnected, the event comes

ESP_GATTC_READ_MULTIPLE_EVT = 42

When the ble characteristic or descriptor multiple complete, the event comes

ESP_GATTC_QUEUE_FULL_EVT = 43

When the gattc command queue full, the event comes

ESP_GATTC_SET_ASSOC_EVT = 44

When the ble gattc set the associated address complete, the event comes

ESP_GATTC_GET_ADDR_LIST_EVT = 45

When the ble get gattc address list in cache finish, the event comes

ESP_GATTC_DIS_SRVC_CMPL_EVT = 46

When the ble discover service complete, the event comes

BLUFI API

Overview BLUFI is a profile based GATT to config ESP32 WIFI to connect/disconnect AP or setup a softap and etc. Use should concern these things:

1. The event sent from profile. Then you need to do something as the event indicate.

2. Security reference. You can write your own Security functions such as symmetrical encryption/decryption and checksum functions. Even you can define the “Key Exchange/Negotiation” procedure.

Application Example Check `bluetooth/bluedroid/ble` folder in ESP-IDF examples, which contains the following application:

- This is the BLUFI demo. This demo can set ESP32’s wifi to softap/station/softap&station mode and config wifi connections - `bluetooth/bluedroid/ble/blufi`

API Reference

Header File

- `bt/host/bluedroid/api/include/api/esp_blufi_api.h`

Functions

`esp_err_t esp_blufi_register_callbacks (esp_blufi_callbacks_t *callbacks)`

This function is called to receive blufi callback event.

Return ESP_OK - success, other - failed

Parameters

- `[in] callbacks`: callback functions

`esp_err_t esp_blufi_profile_init (void)`

This function is called to initialize blufi_profile.

Return ESP_OK - success, other - failed

`esp_err_t esp_blufi_profile_deinit (void)`

This function is called to de-initialize blufi_profile.

Return ESP_OK - success, other - failed

`esp_err_t esp_blufi_send_wifi_conn_report (wifi_mode_t opmode, esp_blufi_sta_conn_state_t sta_conn_state, uint8_t softap_conn_num, esp_blufi_extra_info_t *extra_info)`

This function is called to send wifi connection report.

Return ESP_OK - success, other - failed

Parameters

- `opmode`: : wifi opmode
- `sta_conn_state`: : station is already in connection or not
- `softap_conn_num`: : softap connection number
- `extra_info`: : extra information, such as `sta_ssid`, `softap_ssid` and etc.

`esp_err_t esp_blufi_send_wifi_list (uint16_t apCount, esp_blufi_ap_record_t *list)`

This function is called to send wifi list.

Return ESP_OK - success, other - failed

Parameters

- `apCount`: : wifi list count
- `list`: : wifi list

`uint16_t esp_blufi_get_version (void)`

Get BLUFI profile version.

Return Most 8bit significant is Great version, Least 8bit is Sub version

`esp_err_t esp_blufi_close (esp_gatt_if_t gatts_if, uint16_t conn_id)`

Close a connection a remote device.

Return

- ESP_OK : success

- other : failed

Parameters

- [in] gatts_if: GATT server access interface
- [in] conn_id: connection ID to be closed.

esp_err_t **esp_blufi_send_error_info** (*esp_blufi_error_state_t* state)

This function is called to send blufi error information.

Return ESP_OK - success, other - failed

Parameters

- state: : error state

esp_err_t **esp_blufi_send_custom_data** (uint8_t *data, uint32_t data_len)

This function is called to custom data.

Return ESP_OK - success, other - failed

Parameters

- data: : custom data value
- data_len: : the length of custom data

Unions

union esp_blufi_cb_param_t

#include <esp_blufi_api.h> BLUFI callback parameters union.

Public Members

struct esp_blufi_cb_param_t::blufi_init_finish_evt_param **init_finish**
Blufi callback param of ESP_BLUFI_EVENT_INIT_FINISH

struct esp_blufi_cb_param_t::blufi_deinit_finish_evt_param **deinit_finish**
Blufi callback param of ESP_BLUFI_EVENT_DEINIT_FINISH

struct esp_blufi_cb_param_t::blufi_set_wifi_mode_evt_param **wifi_mode**
Blufi callback param of ESP_BLUFI_EVENT_INIT_FINISH

struct esp_blufi_cb_param_t::blufi_connect_evt_param **connect**
Blufi callback param of ESP_BLUFI_EVENT_CONNECT

struct esp_blufi_cb_param_t::blufi_disconnect_evt_param **disconnect**
Blufi callback param of ESP_BLUFI_EVENT_DISCONNECT

struct esp_blufi_cb_param_t::blufi_recv_sta_bssid_evt_param **sta_bssid**
Blufi callback param of ESP_BLUFI_EVENT_RECV_STA_BSSID

struct esp_blufi_cb_param_t::blufi_recv_sta_ssid_evt_param **sta_ssid**
Blufi callback param of ESP_BLUFI_EVENT_RECV_STA_SSID

struct esp_blufi_cb_param_t::blufi_recv_sta_passwd_evt_param **sta_passwd**
Blufi callback param of ESP_BLUFI_EVENT_RECV_STA_PASSWD

struct esp_blufi_cb_param_t::blufi_recv_softap_ssid_evt_param **softap_ssid**
Blufi callback param of ESP_BLUFI_EVENT_RECV_SOFTAP_SSID

struct esp_blufi_cb_param_t::blufi_recv_softap_passwd_evt_param **softap_passwd**
Blufi callback param of ESP_BLUFI_EVENT_RECV_SOFTAP_PASSWD

struct esp_blufi_cb_param_t::blufi_recv_softap_max_conn_num_evt_param **softap_max_conn_num**
Blufi callback param of ESP_BLUFI_EVENT_RECV_SOFTAP_MAX_CONN_NUM

struct esp_blufi_cb_param_t::blufi_recv_softap_auth_mode_evt_param **softap_auth_mode**
Blufi callback param of ESP_BLUFI_EVENT_RECV_SOFTAP_AUTH_MODE

struct esp_blufi_cb_param_t::blufi_recv_softap_channel_evt_param **softap_channel**
Blufi callback param of ESP_BLUFI_EVENT_RECV_SOFTAP_CHANNEL

```
struct esp_blufi_cb_param_t::blufi_rcv_username_evt_param username  
    Blufi callback param of ESP_BLUFI_EVENT_RECV_USERNAME  
  
struct esp_blufi_cb_param_t::blufi_rcv_ca_evt_param ca  
    Blufi callback param of ESP_BLUFI_EVENT_RECV_CA_CERT  
  
struct esp_blufi_cb_param_t::blufi_rcv_client_cert_evt_param client_cert  
    Blufi callback param of ESP_BLUFI_EVENT_RECV_CLIENT_CERT  
  
struct esp_blufi_cb_param_t::blufi_rcv_server_cert_evt_param server_cert  
    Blufi callback param of ESP_BLUFI_EVENT_RECV_SERVER_CERT  
  
struct esp_blufi_cb_param_t::blufi_rcv_client_pkey_evt_param client_pkey  
    Blufi callback param of ESP_BLUFI_EVENT_RECV_CLIENT_PRIV_KEY  
  
struct esp_blufi_cb_param_t::blufi_rcv_server_pkey_evt_param server_pkey  
    Blufi callback param of ESP_BLUFI_EVENT_RECV_SERVER_PRIV_KEY  
  
struct esp_blufi_cb_param_t::blufi_get_error_evt_param report_error  
    Blufi callback param of ESP_BLUFI_EVENT_REPORT_ERROR  
  
struct esp_blufi_cb_param_t::blufi_rcv_custom_data_evt_param custom_data  
    Blufi callback param of ESP_BLUFI_EVENT_RECV_CUSTOM_DATA  
  
struct blufi_connect_evt_param  
    #include <esp_blufi_api.h> ESP_BLUFI_EVENT_CONNECT.
```

Public Members

esp_bd_addr_t **remote_bda**
 Blufi Remote bluetooth device address

uint8_t **server_if**
 server interface

uint16_t **conn_id**
 Connection id

```
struct blufi_deinit_finish_evt_param  
    #include <esp_blufi_api.h> ESP_BLUFI_EVENT_DEINIT_FINISH.
```

Public Members

esp_blufi_deinit_state_t **state**
 De-initial status

```
struct blufi_disconnect_evt_param  
    #include <esp_blufi_api.h> ESP_BLUFI_EVENT_DISCONNECT.
```

Public Members

esp_bd_addr_t **remote_bda**
 Blufi Remote bluetooth device address

```
struct blufi_get_error_evt_param  
    #include <esp_blufi_api.h> ESP_BLUFI_EVENT_REPORT_ERROR.
```

Public Members

esp_blufi_error_state_t **state**
Blufi error state

struct blufi_init_finish_evt_param
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_INIT_FINISH.

Public Members

esp_blufi_init_state_t **state**
Initial status

struct blufi_recv_ca_evt_param
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_CA_CERT.

Public Members

uint8_t ***cert**
CA certificate point

int **cert_len**
CA certificate length

struct blufi_recv_client_cert_evt_param
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_CLIENT_CERT

Public Members

uint8_t ***cert**
Client certificate point

int **cert_len**
Client certificate length

struct blufi_recv_client_pkey_evt_param
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_CLIENT_PRIV_KEY

Public Members

uint8_t ***pkey**
Client Private Key point, if Client certificate not contain Key

int **pkey_len**
Client Private key length

struct blufi_recv_custom_data_evt_param
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_CUSTOM_DATA.

Public Members

uint8_t ***data**
Custom data

uint32_t **data_len**
Custom data Length

```
struct blufi_recv_server_cert_evt_param  
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_SERVER_CERT
```

Public Members

uint8_t ***cert**
Client certificate point

int **cert_len**
Client certificate length

```
struct blufi_recv_server_pkey_evt_param  
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_SERVER_PRIV_KEY
```

Public Members

uint8_t ***pkey**
Client Private Key point, if Client certificate not contain Key

int **pkey_len**
Client Private key length

```
struct blufi_recv_softap_auth_mode_evt_param  
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_SOFTAP_AUTH_MODE.
```

Public Members

[*wifi_auth_mode_t*](#) **auth_mode**
Authentication mode

```
struct blufi_recv_softap_channel_evt_param  
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_SOFTAP_CHANNEL.
```

Public Members

uint8_t **channel**
Authentication mode

```
struct blufi_recv_softap_max_conn_num_evt_param  
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_SOFTAP_MAX_CONN_NUM.
```

Public Members

int **max_conn_num**
SSID

```
struct blufi_recv_softap_passwd_evt_param  
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_SOFTAP_PASSWD.
```

Public Members

uint8_t ***passwd**
Password

int **passwd_len**
Password Length


```
struct blufi_rcv_softap_ssid_evt_param  
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_SOFTAP_SSID.
```

Public Members

```
uint8_t *ssid  
SSID  
  
int ssid_len  
SSID length
```

```
struct blufi_rcv_sta_bssid_evt_param  
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_STA_BSSID.
```

Public Members

```
uint8_t bssid[6]  
BSSID
```

```
struct blufi_rcv_sta_passwd_evt_param  
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_STA_PASSWD.
```

Public Members

```
uint8_t *passwd  
Password  
  
int passwd_len  
Password Length
```

```
struct blufi_rcv_sta_ssid_evt_param  
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_STA_SSID.
```

Public Members

```
uint8_t *ssid  
SSID  
  
int ssid_len  
SSID length
```

```
struct blufi_rcv_username_evt_param  
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_USERNAME.
```

Public Members

```
uint8_t *name  
Username point  
  
int name_len  
Username length
```

```
struct blufi_set_wifi_mode_evt_param  
#include <esp_blufi_api.h> ESP_BLUFI_EVENT_SET_WIFI_MODE.
```

Public Members

wifi_mode_t **op_mode**
Wifi operation mode

Structures

struct esp_blufi_extra_info_t
BLUFI extra information structure.

Public Members

uint8_t **sta_bssid**[6]
BSSID of station interface

bool **sta_bssid_set**
is BSSID of station interface set

uint8_t ***sta_ssid**
SSID of station interface

int **sta_ssid_len**
length of SSID of station interface

uint8_t ***sta_passwd**
password of station interface

int **sta_passwd_len**
length of password of station interface

uint8_t ***softap_ssid**
SSID of softap interface

int **softap_ssid_len**
length of SSID of softap interface

uint8_t ***softap_passwd**
password of station interface

int **softap_passwd_len**
length of password of station interface

uint8_t **softap_authmode**
authentication mode of softap interface

bool **softap_authmode_set**
is authentication mode of softap interface set

uint8_t **softap_max_conn_num**
max connection number of softap interface

bool **softap_max_conn_num_set**
is max connection number of softap interface set

uint8_t **softap_channel**
channel of softap interface

bool **softap_channel_set**
is channel of softap interface set

struct esp_blufi_ap_record_t
Description of an WiFi AP.

Public Members

`uint8_t ssid[33]`
SSID of AP

`int8_t rssi`
signal strength of AP

struct esp_blufi_callbacks_t
BLUFI callback functions type.

Public Members

esp_blufi_event_cb_t **event_cb**
BLUFI event callback

esp_blufi_negotiate_data_handler_t **negotiate_data_handler**
BLUFI negotiate data function for negotiate share key

esp_blufi_encrypt_func_t **encrypt_func**
BLUFI encrypt data function with share key generated by negotiate_data_handler

esp_blufi_decrypt_func_t **decrypt_func**
BLUFI decrypt data function with share key generated by negotiate_data_handler

esp_blufi_checksum_func_t **checksum_func**
BLUFI check sum function (FCS)

Type Definitions

typedef void (*esp_blufi_event_cb_t) (*esp_blufi_cb_event_t* event, *esp_blufi_cb_param_t* *param)
BLUFI event callback function type.

Parameters

- event: : Event type
- param: : Point to callback parameter, currently is union type

typedef void (*esp_blufi_negotiate_data_handler_t) (uint8_t *data, int len, uint8_t **output_data, int *output_len, bool *need_free)

BLUFI negotiate data handler.

Parameters

- data: : data from phone
- len: : length of data from phone
- output_data: : data want to send to phone
- output_len: : length of data want to send to phone
- need_free: : output reporting if memory needs to be freed or not *

typedef int (*esp_blufi_encrypt_func_t) (uint8_t iv8, uint8_t *crypt_data, int crypt_len)
BLUFI encrypt the data after negotiate a share key.

Return Nonnegative number is encrypted length, if error, return negative number;

Parameters

- iv8: : initial vector(8bit), normally, blufi core will input packet sequence number
- crypt_data: : plain text and encrypted data, the encrypt function must support autochthonous encrypt
- crypt_len: : length of plain text

typedef int (*esp_blufi_decrypt_func_t) (uint8_t iv8, uint8_t *crypt_data, int crypt_len)
BLUFI decrypt the data after negotiate a share key.

Return Nonnegative number is decrypted length, if error, return negative number;

Parameters

- `iv8`: : initial vector(8bit), normally, blufi core will input packet sequence number
- `crypt_data`: : encrypted data and plain text, the encrypt function must support autochthonous decrypt
- `crypt_len`: : length of encrypted text

```
typedef uint16_t(*esp_blufi_checksum_func_t)(uint8_t iv8, uint8_t *data, int len)
BLUFI checksum.
```

Parameters

- `iv8`: : initial vector(8bit), normally, blufi core will input packet sequence number
- `data`: : data need to checksum
- `len`: : length of data

Enumerations

```
enum esp_blufi_cb_event_t
```

Values:

```
ESP_BLUFI_EVENT_INIT_FINISH = 0
ESP_BLUFI_EVENT_DEINIT_FINISH
ESP_BLUFI_EVENT_SET_WIFI_OPMODE
ESP_BLUFI_EVENT_BLE_CONNECT
ESP_BLUFI_EVENT_BLE_DISCONNECT
ESP_BLUFI_EVENT_REQ_CONNECT_TO_AP
ESP_BLUFI_EVENT_REQ_DISCONNECT_FROM_AP
ESP_BLUFI_EVENT_GET_WIFI_STATUS
ESP_BLUFI_EVENT_DEAUTHENTICATE_STA
ESP_BLUFI_EVENT_RECV_STA_BSSID
ESP_BLUFI_EVENT_RECV_STA_SSID
ESP_BLUFI_EVENT_RECV_STA_PASSWD
ESP_BLUFI_EVENT_RECV_SOFTAP_SSID
ESP_BLUFI_EVENT_RECV_SOFTAP_PASSWD
ESP_BLUFI_EVENT_RECV_SOFTAP_MAX_CONN_NUM
ESP_BLUFI_EVENT_RECV_SOFTAP_AUTH_MODE
ESP_BLUFI_EVENT_RECV_SOFTAP_CHANNEL
ESP_BLUFI_EVENT_RECV_USERNAME
ESP_BLUFI_EVENT_RECV_CA_CERT
ESP_BLUFI_EVENT_RECV_CLIENT_CERT
ESP_BLUFI_EVENT_RECV_SERVER_CERT
ESP_BLUFI_EVENT_RECV_CLIENT_PRIV_KEY
ESP_BLUFI_EVENT_RECV_SERVER_PRIV_KEY
ESP_BLUFI_EVENT_RECV_SLAVE_DISCONNECT_BLE
ESP_BLUFI_EVENT_GET_WIFI_LIST
ESP_BLUFI_EVENT_REPORT_ERROR
ESP_BLUFI_EVENT_RECV_CUSTOM_DATA
```

enum esp_blufi_sta_conn_state_t

BLUFI config status.

Values:

ESP_BLUFI_STA_CONN_SUCCESS = 0x00

ESP_BLUFI_STA_CONN_FAIL = 0x01

enum esp_blufi_init_state_t

BLUFI init status.

Values:

ESP_BLUFI_INIT_OK = 0

ESP_BLUFI_INIT_FAILED

enum esp_blufi_deinit_state_t

BLUFI deinit status.

Values:

ESP_BLUFI_DEINIT_OK = 0

ESP_BLUFI_DEINIT_FAILED

enum esp_blufi_error_state_t

Values:

ESP_BLUFI_SEQUENCE_ERROR = 0

ESP_BLUFI_CHECKSUM_ERROR

ESP_BLUFI_DECRYPT_ERROR

ESP_BLUFI_ENCRYPT_ERROR

ESP_BLUFI_INIT_SECURITY_ERROR

ESP_BLUFI_DH_MALLOC_ERROR

ESP_BLUFI_DH_PARAM_ERROR

ESP_BLUFI_READ_PARAM_ERROR

ESP_BLUFI_MAKE_PUBLIC_ERROR

ESP_BLUFI_DATA_FORMAT_ERROR

2.1.4 CLASSIC BT

CLASSIC BLUETOOTH GAP API

Overview [Instructions](#)

Application Example [Instructions](#)

API Reference

Header File

- [bt/host/bluedroid/api/include/api/esp_gap_bt_api.h](#)

Functions

static uint32_t **esp_bt_gap_get_cod_srvc** (uint32_t *cod*)
get major service field of COD

Return major service bits

Parameters

- [in] *cod*: Class of Device

static uint32_t **esp_bt_gap_get_cod_major_dev** (uint32_t *cod*)
get major device field of COD

Return major device bits

Parameters

- [in] *cod*: Class of Device

static uint32_t **esp_bt_gap_get_cod_minor_dev** (uint32_t *cod*)
get minor service field of COD

Return minor service bits

Parameters

- [in] *cod*: Class of Device

static uint32_t **esp_bt_gap_get_cod_format_type** (uint32_t *cod*)
get format type of COD

Return format type

Parameters

- [in] *cod*: Class of Device

static bool **esp_bt_gap_is_valid_cod** (uint32_t *cod*)
decide the integrity of COD

Return

- true if *cod* is valid
- false otherwise

Parameters

- [in] *cod*: Class of Device

esp_err_t **esp_bt_gap_register_callback** (*esp_bt_gap_cb_t* *callback*)
register callback function. This function should be called after `esp_bluedroid_enable()` completes successfully

Return

- ESP_OK : Succeed
- ESP_FAIL: others

esp_err_t **esp_bt_gap_set_scan_mode** (*esp_bt_connection_mode_t* *c_mode*,
esp_bt_discovery_mode_t *d_mode*)

Set discoverability and connectability mode for legacy bluetooth. This function should be called after `esp_bluedroid_enable()` completes successfully.

Return

- ESP_OK : Succeed
- ESP_ERR_INVALID_ARG: if argument invalid
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- [in] *c_mode*: : one of the enums of `esp_bt_connection_mode_t`
- [in] *d_mode*: : one of the enums of `esp_bt_discovery_mode_t`

esp_err_t **esp_bt_gap_start_discovery** (*esp_bt_inq_mode_t* *mode*, uint8_t *inq_len*, uint8_t
num_rsps)

This function starts Inquiry and Name Discovery. It should be called after `esp_bluedroid_enable()` completes successfully. When Inquiry is halted and cached results do not contain device name, then Name Discovery will connect to the peer target to get the device name. `esp_bt_gap_cb_t` will be called with `ESP_BT_GAP_DISC_STATE_CHANGED_EVT` when Inquiry is started or Name Discovery is completed.

esp_bt_gap_cb_t will be called with ESP_BT_GAP_DISC_RES_EVT each time the two types of discovery results are got.

Return

- ESP_OK : Succeed
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_ERR_INVALID_ARG: if invalid parameters are provided
- ESP_FAIL: others

Parameters

- [in] mode: - Inquiry mode
- [in] inq_len: - Inquiry duration in 1.28 sec units, ranging from 0x01 to 0x30. This parameter only specifies the total duration of the Inquiry process,
 - when this time expires, Inquiry will be halted.
- [in] num_rsps: - Number of responses that can be received before the Inquiry is halted, value 0 indicates an unlimited number of responses.

esp_err_t esp_bt_gap_cancel_discovery (void)

Cancel Inquiry and Name Discovery. This function should be called after esp_bluedroid_enable() completes successfully. esp_bt_gap_cb_t will be called with ESP_BT_GAP_DISC_STATE_CHANGED_EVT if Inquiry or Name Discovery is cancelled by calling this function.

Return

- ESP_OK : Succeed
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t esp_bt_gap_get_remote_services (esp_bd_addr_t remote_bda)

Start SDP to get remote services. This function should be called after esp_bluedroid_enable() completes successfully. esp_bt_gap_cb_t will be called with ESP_BT_GAP_RMT_SRVCS_EVT after service discovery ends.

Return

- ESP_OK : Succeed
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t esp_bt_gap_get_remote_service_record (esp_bd_addr_t remote_bda, esp_bt_uuid_t *uuid)

Start SDP to look up the service matching uuid on the remote device. This function should be called after esp_bluedroid_enable() completes successfully.

esp_bt_gap_cb_t will be called with ESP_BT_GAP_RMT_SRVC_REC_EVT after service discovery ends

Return

- ESP_OK : Succeed
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

uint8_t *esp_bt_gap_resolve_eir_data (uint8_t *eir, esp_bt_eir_type_t type, uint8_t *length)

This function is called to get EIR data for a specific type.

Return pointer of starting position of eir data excluding eir data type, NULL if not found

Parameters

- [in] eir: - pointer of raw eir data to be resolved
- [in] type: - specific EIR data type
- [out] length: - return the length of EIR data excluding fields of length and data type

esp_err_t esp_bt_gap_config_eir_data (esp_bt_eir_data_t *eir_data)

This function is called to config EIR data.

esp_bt_gap_cb_t will be called with ESP_BT_GAP_CONFIG_EIR_DATA_EVT after config EIR ends.

Return

- ESP_OK : Succeed
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled

- ESP_ERR_INVALID_ARG: if param is invalid
- ESP_FAIL: others

Parameters

- [in] eir_data: - pointer of EIR data content

esp_err_t **esp_bt_gap_set_cod** (*esp_bt_cod_t* cod, *esp_bt_cod_mode_t* mode)

This function is called to set class of device. esp_bt_gap_cb_t will be called with ESP_BT_GAP_SET_COD_EVT after set COD ends. Some profile have special restrictions on class of device, changes may cause these profile do not work.

Return

- ESP_OK : Succeed
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_ERR_INVALID_ARG: if param is invalid
- ESP_FAIL: others

Parameters

- [in] cod: - class of device
- [in] mode: - setting mode

esp_err_t **esp_bt_gap_get_cod** (*esp_bt_cod_t* *cod)

This function is called to get class of device.

Return

- ESP_OK : Succeed
- ESP_FAIL: others

Parameters

- [out] cod: - class of device

esp_err_t **esp_bt_gap_read_rssi_delta** (*esp_bd_addr_t* remote_addr)

This function is called to read RSSI delta by address after connected. The RSSI value returned by ESP_BT_GAP_READ_RSSI_DELTA_EVT.

Return

- ESP_OK : Succeed
- ESP_FAIL: others

Parameters

- [in] remote_addr: - remote device address, corresponding to a certain connection handle.

esp_err_t **esp_bt_gap_remove_bond_device** (*esp_bd_addr_t* bd_addr)

Removes a device from the security database list of peer device.

Return - ESP_OK : success

- ESP_FAIL : failed

Parameters

- [in] bd_addr: : BD address of the peer device

int **esp_bt_gap_get_bond_device_num** (void)

Get the device number from the security database list of peer device. It will return the device bonded number immediately.

Return - >= 0 : bonded devices number.

- ESP_FAIL : failed

esp_err_t **esp_bt_gap_get_bond_device_list** (int *dev_num, *esp_bd_addr_t* *dev_list)

Get the device from the security database list of peer device. It will return the device bonded information immediately.

Return

- ESP_OK : Succeed
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- [inout] dev_num: Indicate the dev_list array(buffer) size as input. If dev_num is large enough, it means the actual number as output. Suggest that dev_num value equal to

`esp_ble_get_bond_device_num()`.

- [out] `dev_list`: an array(buffer) of `esp_bd_addr_t` type. Use for storing the bonded devices address. The `dev_list` should be allocated by who call this API.

esp_err_t **esp_bt_gap_set_pin** (*esp_bt_pin_type_t* pin_type, *uint8_t* pin_code_len, *esp_bt_pin_code_t* pin_code)

Set pin type and default pin code for legacy pairing.

Return - `ESP_OK` : success

- `ESP_ERR_INVALID_STATE`: if bluetooth stack is not yet enabled
- other : failed

Parameters

- [in] `pin_type`: Use variable or fixed pin. If `pin_type` is `ESP_BT_PIN_TYPE_VARIABLE`, `pin_code` and `pin_code_len` will be ignored, and `ESP_BT_GAP_PIN_REQ_EVT` will come when control requests for pin code. Else, will use fixed pin code and not callback to users.
- [in] `pin_code_len`: Length of `pin_code`
- [in] `pin_code`: Pin_code

esp_err_t **esp_bt_gap_pin_reply** (*esp_bd_addr_t* bd_addr, *bool* accept, *uint8_t* pin_code_len, *esp_bt_pin_code_t* pin_code)

Reply the `pin_code` to the peer device for legacy pairing when `ESP_BT_GAP_PIN_REQ_EVT` is coming.

Return - `ESP_OK` : success

- `ESP_ERR_INVALID_STATE`: if bluetooth stack is not yet enabled
- other : failed

Parameters

- [in] `bd_addr`: BD address of the peer
- [in] `accept`: Pin_code reply successful or declined.
- [in] `pin_code_len`: Length of `pin_code`
- [in] `pin_code`: Pin_code

esp_err_t **esp_bt_gap_set_security_param** (*esp_bt_sp_param_t* param_type, *void* *value, *uint8_t* len)

Set a GAP security parameter value. Overrides the default value.

Return - `ESP_OK` : success

- `ESP_ERR_INVALID_STATE`: if bluetooth stack is not yet enabled
- other : failed

Parameters

- [in] `param_type`: : the type of the param which is to be set
- [in] `value`: : the param value
- [in] `len`: : the length of the param value

esp_err_t **esp_bt_gap_ssp_passkey_reply** (*esp_bd_addr_t* bd_addr, *bool* accept, *uint32_t* passkey)

Reply the key value to the peer device in the legacy connection stage.

Return - `ESP_OK` : success

- `ESP_ERR_INVALID_STATE`: if bluetooth stack is not yet enabled
- other : failed

Parameters

- [in] `bd_addr`: : BD address of the peer
- [in] `accept`: : passkey entry successful or declined.
- [in] `passkey`: : passkey value, must be a 6 digit number, can be lead by 0.

esp_err_t **esp_bt_gap_ssp_confirm_reply** (*esp_bd_addr_t* bd_addr, *bool* accept)

Reply the confirm value to the peer device in the legacy connection stage.

Return - `ESP_OK` : success

- `ESP_ERR_INVALID_STATE`: if bluetooth stack is not yet enabled
- other : failed

Parameters

- [in] `bd_addr`: : BD address of the peer device
- [in] `accept`: : numbers to compare are the same or different.

esp_err_t **esp_bt_gap_set_afh_channels** (*esp_bt_gap_afh_channels* channels)

Set the AFH channels.

Return - ESP_OK : success

- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- other : failed

Parameters

- [in] channels: : The nth such field (in the range 0 to 78) contains the value for channel n : 0 means channel n is bad. 1 means channel n is unknown. The most significant bit is reserved and shall be set to 0. At least 20 channels shall be marked as unknown.

esp_err_t **esp_bt_gap_read_remote_name** (*esp_bd_addr_t* remote_bda)

Read the remote device name.

Return - ESP_OK : success

- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- other : failed

Parameters

- [in] remote_bda: The remote device' s address

esp_err_t **esp_bt_gap_set_qos** (*esp_bd_addr_t* remote_bda, *uint32_t* t_poll)

Config Quality of service.

Return - ESP_OK : success

- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- other : failed

Parameters

- [in] remote_bda: The remote device' s address
- [in] t_poll: Poll interval, the maximum time between transmissions which from the master to a particular slave on the ACL logical transport. unit is 0.625ms

Unions

union **esp_bt_gap_cb_param_t**

#include <esp_gap_bt_api.h> A2DP state callback parameters.

Public Members

struct *esp_bt_gap_cb_param_t::disc_res_param* **disc_res**

discovery result parameter struct

struct *esp_bt_gap_cb_param_t::disc_state_changed_param* **disc_st_chg**

discovery state changed parameter struct

struct *esp_bt_gap_cb_param_t::rmt_srvcs_param* **rmt_srvcs**

services of remote device parameter struct

struct *esp_bt_gap_cb_param_t::rmt_srvc_rec_param* **rmt_srvc_rec**

specific service record from remote device parameter struct

struct *esp_bt_gap_cb_param_t::read_rssi_delta_param* **read_rssi_delta**

read rssi parameter struct

struct *esp_bt_gap_cb_param_t::config_eir_data_param* **config_eir_data**

config EIR data

struct *esp_bt_gap_cb_param_t::auth_cmpl_param* **auth_cmpl**

authentication complete parameter struct

struct *esp_bt_gap_cb_param_t::pin_req_param* **pin_req**

pin request parameter struct

struct *esp_bt_gap_cb_param_t::cfm_req_param* **cfm_req**

confirm request parameter struct

```
struct esp_bt_gap_cb_param_t::key_notif_param key_notif  
    passkey notif parameter struct  
  
struct esp_bt_gap_cb_param_t::key_req_param key_req  
    passkey request parameter struct  
  
struct esp_bt_gap_cb_param_t::set_afh_channels_param set_afh_channels  
    set AFH channel parameter struct  
  
struct esp_bt_gap_cb_param_t::read_rmt_name_param read_rmt_name  
    read Remote Name parameter struct  
  
struct esp_bt_gap_cb_param_t::bt_remove_bond_dev_cmpl_evt_param remove_bond_dev_cmpl  
    Event parameter of ESP_BT_GAP_REMOVE_BOND_DEV_COMPLETE_EVT  
  
struct esp_bt_gap_cb_param_t::qos_cmpl_param qos_cmpl  
    QoS complete parameter struct  
  
struct auth_cmpl_param  
    #include <esp_gap_bt_api.h> ESP_BT_GAP_AUTH_CMPL_EVT.
```

Public Members

```
esp_bd_addr_t bda  
    remote bluetooth device address  
  
esp_bt_status_t stat  
    authentication complete status  
  
uint8_t device_name[ESP_BT_GAP_MAX_BDNAME_LEN + 1]  
    device name  
  
struct bt_remove_bond_dev_cmpl_evt_param  
    #include <esp_gap_bt_api.h> ESP_BT_GAP_REMOVE_BOND_DEV_COMPLETE_EVT.
```

Public Members

```
esp_bd_addr_t bda  
    remote bluetooth device address  
  
esp_bt_status_t status  
    Indicate the remove bond device operation success status  
  
struct cfm_req_param  
    #include <esp_gap_bt_api.h> ESP_BT_GAP_CFM_REQ_EVT.
```

Public Members

```
esp_bd_addr_t bda  
    remote bluetooth device address  
  
uint32_t num_val  
    the numeric value for comparison.  
  
struct config_eir_data_param  
    #include <esp_gap_bt_api.h> ESP_BT_GAP_CONFIG_EIR_DATA_EVT*.
```

Public Members

esp_bt_status_t **stat**

config EIR status: ESP_BT_STATUS_SUCCESS: config success
ESP_BT_STATUS_EIR_TOO_LARGE: the EIR data is more than 240B. The EIR may not contain the whole data. others: failed

uint8_t **eir_type_num**

the number of EIR types in EIR type

esp_bt_eir_type_t **eir_type**[ESP_BT_EIR_TYPE_MAX_NUM]

EIR types in EIR type

struct disc_res_param

#include <esp_gap_bt_api.h> ESP_BT_GAP_DISC_RES_EVT.

Public Members

esp_bd_addr_t **bda**

remote bluetooth device address

int **num_prop**

number of properties got

esp_bt_gap_dev_prop_t ***prop**

properties discovered from the new device

struct disc_state_changed_param

#include <esp_gap_bt_api.h> ESP_BT_GAP_DISC_STATE_CHANGED_EVT.

Public Members

esp_bt_gap_discovery_state_t **state**

discovery state

struct key_notif_param

#include <esp_gap_bt_api.h> ESP_BT_GAP_KEY_NOTIF_EVT.

Public Members

esp_bd_addr_t **bda**

remote bluetooth device address

uint32_t **passkey**

the numeric value for passkey entry.

struct key_req_param

#include <esp_gap_bt_api.h> ESP_BT_GAP_KEY_REQ_EVT.

Public Members

esp_bd_addr_t **bda**

remote bluetooth device address

struct pin_req_param

#include <esp_gap_bt_api.h> ESP_BT_GAP_PIN_REQ_EVT.

Public Members

esp_bd_addr_t **bda**
remote bluetooth device address

bool **min_16_digit**
TRUE if the pin returned must be at least 16 digits

struct qos_cmpl_param
#include <esp_gap_bt_api.h> ESP_BT_GAP_QOS_CMPL_EVT.

Public Members

esp_bt_status_t **stat**
QoS status

esp_bd_addr_t **bda**
remote bluetooth device address

uint32_t **t_poll**
poll interval, the maximum time between transmissions which from the master to a particular slave on the ACL logical transport. unit is 0.625ms.

struct read_rmt_name_param
#include <esp_gap_bt_api.h> ESP_BT_GAP_READ_REMOTE_NAME_EVT.

Public Members

esp_bt_status_t **stat**
read Remote Name status

uint8_t **rmt_name**[ESP_BT_GAP_MAX_BDNAME_LEN + 1]
Remote device name

struct read_rssi_delta_param
#include <esp_gap_bt_api.h> ESP_BT_GAP_READ_RSSI_DELTA_EVT *.

Public Members

esp_bd_addr_t **bda**
remote bluetooth device address

esp_bt_status_t **stat**
read rssi status

int8_t **rssi_delta**
rssi delta value range -128 ~127, The value zero indicates that the RSSI is inside the Golden Receive Power Range, the Golden Receive Power Range is from ESP_BT_GAP_RSSI_LOW_THRLD to ESP_BT_GAP_RSSI_HIGH_THRLD

struct rmt_srvc_rec_param
#include <esp_gap_bt_api.h> ESP_BT_GAP_RMT_SRVC_REC_EVT.

Public Members

esp_bd_addr_t **bda**
remote bluetooth device address

esp_bt_status_t **stat**
service search status

```
struct rmt_srvcs_param
    #include <esp_gap_bt_api.h> ESP_BT_GAP_RMT_SRVCS_EVT.
```

Public Members

esp_bd_addr_t **bda**
remote bluetooth device address

esp_bt_status_t **stat**
service search status

int **num_uuids**
number of UUID in uuid_list

esp_bt_uuid_t ***uuid_list**
list of service UUIDs of remote device

```
struct set_afh_channels_param
    #include <esp_gap_bt_api.h> ESP_BT_GAP_SET_AFH_CHANNELS_EVT.
```

Public Members

esp_bt_status_t **stat**
set AFH channel status

Structures

```
struct esp_bt_cod_t
    Class of device.
```

Public Members

uint32_t **reserved_2** : 2
undefined

uint32_t **minor** : 6
minor class

uint32_t **major** : 5
major class

uint32_t **service** : 11
service class

uint32_t **reserved_8** : 8
undefined

```
struct esp_bt_gap_dev_prop_t
    Bluetooth Device Property Descriptor.
```

Public Members

esp_bt_gap_dev_prop_type_t **type**
device property type

int **len**
device property value length

void ***val**
device property value

struct esp_bt_eir_data_t

EIR data content, according to “Supplement to the Bluetooth Core Specification” .

Public Members**bool fec_required**

FEC is required or not, true by default

bool include_txpower

EIR data include TX power, false by default

bool include_uuid

EIR data include UUID, false by default

uint8_t flag

EIR flags, see ESP_BT_EIR_FLAG for details, EIR will not include flag if it is 0, 0 by default

uint16_t manufacturer_len

Manufacturer data length, 0 by default

uint8_t *p_manufacturer_data

Manufacturer data point

uint16_t url_len

URL length, 0 by default

uint8_t *p_url

URL point

Macros**ESP_BT_GAP_RSSI_HIGH_THRDL**

RSSI threshold.

High RSSI threshold

ESP_BT_GAP_RSSI_LOW_THRDL

Low RSSI threshold

ESP_BT_GAP_AFH_CHANNELS_LEN**ESP_BT_GAP_MAX_BDNAME_LEN**

Maximum bytes of Bluetooth device name.

ESP_BT_GAP_EIR_DATA_LEN

Maximum size of EIR Significant part.

ESP_BT_EIR_TYPE_FLAGS

Extended Inquiry Response data type.

Flag with information such as BR/EDR and LE support

ESP_BT_EIR_TYPE_INCMPL_16BITS_UUID

Incomplete list of 16-bit service UUIDs

ESP_BT_EIR_TYPE_CMPL_16BITS_UUID

Complete list of 16-bit service UUIDs

ESP_BT_EIR_TYPE_INCMPL_32BITS_UUID

Incomplete list of 32-bit service UUIDs

ESP_BT_EIR_TYPE_CMPL_32BITS_UUID

Complete list of 32-bit service UUIDs

ESP_BT_EIR_TYPE_INCMPL_128BITS_UUID

Incomplete list of 128-bit service UUIDs

ESP_BT_EIR_TYPE_CMPL_128BITS_UUID

Complete list of 128-bit service UUIDs

ESP_BT_EIR_TYPE_SHORT_LOCAL_NAME

Shortened Local Name

ESP_BT_EIR_TYPE_CMPL_LOCAL_NAME

Complete Local Name

ESP_BT_EIR_TYPE_TX_POWER_LEVEL

Tx power level, value is 1 octet ranging from -127 to 127, unit is dBm

ESP_BT_EIR_TYPE_URL

Uniform resource identifier

ESP_BT_EIR_TYPE_MANU_SPECIFIC

Manufacturer specific data

ESP_BT_EIR_TYPE_MAX_NUM

MAX number of EIR type

ESP_BT_EIR_FLAG_LIMIT_DISC

ESP_BT_EIR_FLAG_GEN_DISC

ESP_BT_EIR_FLAG_BREDR_NOT_SPT

ESP_BT_EIR_FLAG_DMT_CONTROLLER_SPT

ESP_BT_EIR_FLAG_DMT_HOST_SPT

ESP_BT_EIR_MAX_LEN

ESP_BT_PIN_CODE_LEN

Max pin code length

ESP_BT_IO_CAP_OUT

ESP_BT_IO_CAP_IO

ESP_BT_IO_CAP_IN

ESP_BT_IO_CAP_NONE

ESP_BT_COD_SRVC_BIT_MASK

Bits of major service class field.

Major service bit mask

ESP_BT_COD_SRVC_BIT_OFFSET

Major service bit offset

ESP_BT_COD_MAJOR_DEV_BIT_MASK

Bits of major device class field.

Major device bit mask

ESP_BT_COD_MAJOR_DEV_BIT_OFFSET

Major device bit offset

ESP_BT_COD_MINOR_DEV_BIT_MASK

Bits of minor device class field.

Minor device bit mask

ESP_BT_COD_MINOR_DEV_BIT_OFFSET

Minor device bit offset

ESP_BT_COD_FORMAT_TYPE_BIT_MASK

Bits of format type.

Format type bit mask

ESP_BT_COD_FORMAT_TYPE_BIT_OFFSET

Format type bit offset

ESP_BT_COD_FORMAT_TYPE_1

Class of device format type 1.

ESP_BT_GAP_MIN_INQ_LEN

Minimum and Maximum inquiry length Minimum inquiry duration, unit is 1.28s

ESP_BT_GAP_MAX_INQ_LEN

Maximum inquiry duration, unit is 1.28s

Type Definitions

```
typedef uint8_t esp_bt_gap_afh_channels[ESP_BT_GAP_AFH_CHANNELS_LEN]
```

```
typedef uint8_t esp_bt_eir_type_t
```

```
typedef uint8_t esp_bt_pin_code_t[ESP_BT_PIN_CODE_LEN]
```

Pin Code (upto 128 bits) MSB is 0

```
typedef uint8_t esp_bt_io_cap_t
```

combination of the io capability

```
typedef void (*esp_bt_gap_cb_t) (esp_bt_gap_cb_event_t event, esp_bt_gap_cb_param_t *param)
```

bluetooth GAP callback function type

Parameters

- event: : Event type
- param: : Pointer to callback parameter

Enumerations

```
enum esp_bt_cod_mode_t
```

class of device settings

Values:

```
ESP_BT_SET_COD_MAJOR_MINOR = 0x01
```

overwrite major, minor class

```
ESP_BT_SET_COD_SERVICE_CLASS = 0x02
```

set the bits in the input, the current bit will remain

```
ESP_BT_CLR_COD_SERVICE_CLASS = 0x04
```

clear the bits in the input, others will remain

```
ESP_BT_SET_COD_ALL = 0x08
```

overwrite major, minor, set the bits in service class

```
ESP_BT_INIT_COD = 0x0a
```

overwrite major, minor, and service class

```
enum esp_bt_connection_mode_t
```

Discoverability and Connectability mode.

Values:

```
ESP_BT_NON_CONNECTABLE
```

Non-connectable

```
ESP_BT_CONNECTABLE
```

Connectable

```
enum esp_bt_discovery_mode_t
```

Values:

```
ESP_BT_NON_DISCOVERABLE
```

Non-discoverable

ESP_BT_LIMITED_DISCOVERABLE
Limited Discoverable

ESP_BT_GENERAL_DISCOVERABLE
General Discoverable

enum esp_bt_gap_dev_prop_type_t
Bluetooth Device Property type.

Values:

ESP_BT_GAP_DEV_PROP_BDNAME = 1
Bluetooth device name, value type is int8_t []

ESP_BT_GAP_DEV_PROP_COD
Class of Device, value type is uint32_t

ESP_BT_GAP_DEV_PROP_RSSI
Received Signal strength Indication, value type is int8_t, ranging from -128 to 127

ESP_BT_GAP_DEV_PROP_EIR
Extended Inquiry Response, value type is uint8_t []

enum esp_bt_cod_srvc_t
Major service class field of Class of Device, mutiple bits can be set.

Values:

ESP_BT_COD_SRVC_NONE = 0
None indicates an invalid value

ESP_BT_COD_SRVC_LMTD_DISCOVER = 0x1
Limited Discoverable Mode

ESP_BT_COD_SRVC_POSITIONING = 0x8
Positioning (Location identification)

ESP_BT_COD_SRVC_NETWORKING = 0x10
Networking, e.g. LAN, Ad hoc

ESP_BT_COD_SRVC_RENDERING = 0x20
Rendering, e.g. Printing, Speakers

ESP_BT_COD_SRVC_CAPTURING = 0x40
Capturing, e.g. Scanner, Microphone

ESP_BT_COD_SRVC_OBJ_TRANSFER = 0x80
Object Transfer, e.g. v-Inbox, v-Folder

ESP_BT_COD_SRVC_AUDIO = 0x100
Audio, e.g. Speaker, Microphone, Headset service

ESP_BT_COD_SRVC_TELEPHONY = 0x200
Telephony, e.g. Cordless telephony, Modem, Headset service

ESP_BT_COD_SRVC_INFORMATION = 0x400
Information, e.g., WEB-server, WAP-server

enum esp_bt_pin_type_t

Values:

ESP_BT_PIN_TYPE_VARIABLE = 0
Refer to BTM_PIN_TYPE_VARIABLE

ESP_BT_PIN_TYPE_FIXED = 1
Refer to BTM_PIN_TYPE_FIXED

enum esp_bt_sp_param_t

Values:

ESP_BT_SP_IOCAP_MODE = 0
Set IO mode

enum esp_bt_cod_major_dev_t
Major device class field of Class of Device.

Values:

ESP_BT_COD_MAJOR_DEV_MISC = 0
Miscellaneous

ESP_BT_COD_MAJOR_DEV_COMPUTER = 1
Computer

ESP_BT_COD_MAJOR_DEV_PHONE = 2
Phone(cellular, cordless, pay phone, modem)

ESP_BT_COD_MAJOR_DEV_LAN_NAP = 3
LAN, Network Access Point

ESP_BT_COD_MAJOR_DEV_AV = 4
Audio/Video(headset, speaker, stereo, video display, VCR)

ESP_BT_COD_MAJOR_DEV_PERIPHERAL = 5
Peripheral(mouse, joystick, keyboard)

ESP_BT_COD_MAJOR_DEV_IMAGING = 6
Imaging(printer, scanner, camera, display)

ESP_BT_COD_MAJOR_DEV_WEARABLE = 7
Wearable

ESP_BT_COD_MAJOR_DEV_TOY = 8
Toy

ESP_BT_COD_MAJOR_DEV_HEALTH = 9
Health

ESP_BT_COD_MAJOR_DEV_UNCATEGORIZED = 31
Uncategorized: device not specified

enum esp_bt_gap_discovery_state_t
Bluetooth Device Discovery state

Values:

ESP_BT_GAP_DISCOVERY_STOPPED
device discovery stopped

ESP_BT_GAP_DISCOVERY_STARTED
device discovery started

enum esp_bt_gap_cb_event_t
BT GAP callback events.

Values:

ESP_BT_GAP_DISC_RES_EVT = 0
device discovery result event

ESP_BT_GAP_DISC_STATE_CHANGED_EVT
discovery state changed event

ESP_BT_GAP_RMT_SRVCS_EVT
get remote services event

ESP_BT_GAP_RMT_SRVC_REC_EVT
get remote service record event

ESP_BT_GAP_AUTH_CMPL_EVT
AUTH complete event

ESP_BT_GAP_PIN_REQ_EVT
Legacy Pairing Pin code request

ESP_BT_GAP_CFM_REQ_EVT
Simple Pairing User Confirmation request.

ESP_BT_GAP_KEY_NOTIF_EVT
Simple Pairing Passkey Notification

ESP_BT_GAP_KEY_REQ_EVT
Simple Pairing Passkey request

ESP_BT_GAP_READ_RSSI_DELTA_EVT
read rssi event

ESP_BT_GAP_CONFIG_EIR_DATA_EVT
config EIR data event

ESP_BT_GAP_SET_AFH_CHANNELS_EVT
set AFH channels event

ESP_BT_GAP_READ_REMOTE_NAME_EVT
read Remote Name event

ESP_BT_GAP_REMOVE_BOND_DEV_COMPLETE_EVT
remove bond device complete event

ESP_BT_GAP_QOS_CMPL_EVT
QOS complete event

ESP_BT_GAP_EVT_MAX

enum esp_bt_inq_mode_t
Inquiry Mode

Values:

ESP_BT_INQ_MODE_GENERAL_INQUIRY
General inquiry mode

ESP_BT_INQ_MODE_LIMITED_INQUIRY
Limited inquiry mode

Bluetooth A2DP API

Overview [Instructions](#)

Application Example Check [bluetooth/bluedroid/classic_bt](#) folder in ESP-IDF examples, which contains the following application:

- This is a A2DP sink client demo. This demo can be discovered and connected by A2DP source device and receive the audio stream from remote device - [bluetooth/bluedroid/classic_bt/a2dp_sink](#)

API Reference

Header File

- [bt/host/bluedroid/api/include/api/esp_a2dp_api.h](#)

Functions

esp_err_t **esp_a2d_register_callback** (*esp_a2d_cb_t* callback)

Register application callback function to A2DP module. This function should be called only after `esp_bluedroid_enable()` completes successfully, used by both A2DP source and sink.

Return

- ESP_OK: success
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: if callback is a NULL function pointer

Parameters

- [in] callback: A2DP event callback function

esp_err_t **esp_a2d_sink_register_data_callback** (*esp_a2d_sink_data_cb_t* callback)

Register A2DP sink data output function; For now the output is PCM data stream decoded from SBC format. This function should be called only after `esp_bluedroid_enable()` completes successfully, used only by A2DP sink. The callback is invoked in the context of A2DP sink task whose stack size is configurable through `menuconfig`.

Return

- ESP_OK: success
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: if callback is a NULL function pointer

Parameters

- [in] callback: A2DP sink data callback function

esp_err_t **esp_a2d_sink_init** (void)

Initialize the bluetooth A2DP sink module. This function should be called after `esp_bluedroid_enable()` completes successfully, and ESP_A2D_PROF_STATE_EVT with ESP_A2D_INIT_SUCCESS will be reported to the APP layer. Note: A2DP can work independently. If you want to use AVRC together, you should initiate AVRC first.

Return

- ESP_OK: if the initialization request is sent successfully
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_a2d_sink_deinit** (void)

De-initialize for A2DP sink module. This function should be called only after `esp_bluedroid_enable()` completes successfully, and ESP_A2D_PROF_STATE_EVT with ESP_A2D_DEINIT_SUCCESS will be reported to APP layer.

Return

- ESP_OK: success
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_a2d_sink_connect** (*esp_bd_addr_t* remote_bda)

Connect to remote bluetooth A2DP source device, must after `esp_a2d_sink_init()`

Return

- ESP_OK: connect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- [in] remote_bda: remote bluetooth device address

esp_err_t **esp_a2d_sink_disconnect** (*esp_bd_addr_t* remote_bda)

Disconnect from the remote A2DP source device.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- [in] `remote_bda`: remote bluetooth device address

esp_err_t **esp_a2d_media_ctrl** (*esp_a2d_media_ctrl_t* *ctrl*)

media control commands; this API can be used for both A2DP sink and source

Return

- `ESP_OK`: control command is sent to lower layer
- `ESP_INVALID_STATE`: if bluetooth stack is not yet enabled
- `ESP_FAIL`: others

Parameters

- [in] *ctrl*: control commands for A2DP data channel

esp_err_t **esp_a2d_source_init** (void)

Initialize the bluetooth A2DP source module. This function should be called after `esp_bluedroid_enable()` completes successfully, and `ESP_A2D_PROF_STATE_EVT` with `ESP_A2D_INIT_SUCCESS` will be reported to the APP layer. Note: A2DP can work independently. If you want to use AVRC together, you should initiate AVRC first.

Return

- `ESP_OK`: if the initialization request is sent successfully
- `ESP_INVALID_STATE`: if bluetooth stack is not yet enabled
- `ESP_FAIL`: others

esp_err_t **esp_a2d_source_deinit** (void)

De-initialize for A2DP source module. This function should be called only after `esp_bluedroid_enable()` completes successfully, and `ESP_A2D_PROF_STATE_EVT` with `ESP_A2D_DEINIT_SUCCESS` will be reported to APP layer.

Return

- `ESP_OK`: success
- `ESP_INVALID_STATE`: if bluetooth stack is not yet enabled
- `ESP_FAIL`: others

esp_err_t **esp_a2d_source_register_data_callback** (*esp_a2d_source_data_cb_t* *callback*)

Register A2DP source data input function; For now the input is PCM data stream. This function should be called only after `esp_bluedroid_enable()` completes successfully. The callback is invoked in the context of A2DP source task whose stack size is configurable through `menuconfig`.

Return

- `ESP_OK`: success
- `ESP_INVALID_STATE`: if bluetooth stack is not yet enabled
- `ESP_FAIL`: if callback is a NULL function pointer

Parameters

- [in] *callback*: A2DP source data callback function

esp_err_t **esp_a2d_source_connect** (*esp_bd_addr_t* *remote_bda*)

Connect to remote A2DP sink device, must after `esp_a2d_source_init()`

Return

- `ESP_OK`: connect request is sent to lower layer
- `ESP_INVALID_STATE`: if bluetooth stack is not yet enabled
- `ESP_FAIL`: others

Parameters

- [in] *remote_bda*: remote bluetooth device address

esp_err_t **esp_a2d_source_disconnect** (*esp_bd_addr_t* *remote_bda*)

Disconnect from the remote A2DP sink device.

Return

- `ESP_OK`: disconnect request is sent to lower layer
- `ESP_INVALID_STATE`: if bluetooth stack is not yet enabled
- `ESP_FAIL`: others

Parameters

- [in] *remote_bda*: remote bluetooth device address

Unions

union esp_a2d_cb_param_t
#include <esp_a2dp_api.h> A2DP state callback parameters.

Public Members

struct esp_a2d_cb_param_t::a2d_conn_stat_param conn_stat
A2DP connection status

struct esp_a2d_cb_param_t::a2d_audio_stat_param audio_stat
audio stream playing state

struct esp_a2d_cb_param_t::a2d_audio_cfg_param audio_cfg
media codec configuration information

struct esp_a2d_cb_param_t::media_ctrl_stat_param media_ctrl_stat
status in acknowledgement to media control commands

struct esp_a2d_cb_param_t::a2d_prof_stat_param a2d_prof_stat
status to indicate a2d prof init or deinit

struct a2d_audio_cfg_param
#include <esp_a2dp_api.h> ESP_A2D_AUDIO_CFG_EVT.

Public Members

esp_bd_addr_t remote_bda
remote bluetooth device address

esp_a2d_mcc_t mcc
A2DP media codec capability information

struct a2d_audio_stat_param
#include <esp_a2dp_api.h> ESP_A2D_AUDIO_STATE_EVT.

Public Members

esp_a2d_audio_state_t state
one of the values from esp_a2d_audio_state_t

esp_bd_addr_t remote_bda
remote bluetooth device address

struct a2d_conn_stat_param
#include <esp_a2dp_api.h> ESP_A2D_CONNECTION_STATE_EVT.

Public Members

esp_a2d_connection_state_t state
one of values from esp_a2d_connection_state_t

esp_bd_addr_t remote_bda
remote bluetooth device address

esp_a2d_disc_rsn_t disc_rsn
reason of disconnection for “DISCONNECTED”

struct a2d_prof_stat_param
#include <esp_a2dp_api.h> ESP_A2D_PROF_STATE_EVT.

Public Members

esp_a2d_init_state_t **init_state**
a2dp profile state param

struct media_ctrl_stat_param
#include <esp_a2dp_api.h> ESP_A2D_MEDIA_CTRL_ACK_EVT.

Public Members

esp_a2d_media_ctrl_t **cmd**
media control commands to acknowledge

esp_a2d_media_ctrl_ack_t **status**
acknowledgement to media control commands

Structures

struct esp_a2d_mcc_t
A2DP media codec capabilities union.

Public Members

esp_a2d_mct_t **type**
A2DP media codec type

union *esp_a2d_mcc_t::[anonymous]* **cie**
A2DP codec information element

Macros

ESP_A2D_MCT_SBC
Media codec types supported by A2DP.

SBC

ESP_A2D_MCT_M12
MPEG-1, 2 Audio

ESP_A2D_MCT_M24
MPEG-2, 4 AAC

ESP_A2D_MCT_ATRAC
ATRAC family

ESP_A2D_MCT_NON_A2DP

ESP_A2D_CIE_LEN_SBC

ESP_A2D_CIE_LEN_M12

ESP_A2D_CIE_LEN_M24

ESP_A2D_CIE_LEN_ATRAC

Type Definitions

typedef uint8_t **esp_a2d_mct_t**

typedef void (***esp_a2d_cb_t**) (*esp_a2d_cb_event_t* event, *esp_a2d_cb_param_t* *param)
A2DP profile callback function type.

Parameters

- event: : Event type
- param: : Pointer to callback parameter

typedef void (***esp_a2d_sink_data_cb_t**) (const uint8_t *buf, uint32_t len)
A2DP profile data callback function.

Parameters

- [in] buf: : data received from A2DP source device and is PCM format decoder from SBC decoder; buf references to a static memory block and can be overwritten by upcoming data
- [in] len: : size(in bytes) in buf

typedef int32_t (***esp_a2d_source_data_cb_t**) (uint8_t *buf, int32_t len)
A2DP source data read callback function.

Return size of bytes read successfully, if the argument len is -1, this value is ignored.

Parameters

- [in] buf: : buffer to be filled with PCM data stream from higher layer
- [in] len: : size(in bytes) of data block to be copied to buf. -1 is an indication to user that data buffer shall be flushed

Enumerations

enum **esp_a2d_connection_state_t**
Bluetooth A2DP connection states.

Values:

ESP_A2D_CONNECTION_STATE_DISCONNECTED = 0
connection released

ESP_A2D_CONNECTION_STATE_CONNECTING
connecting remote device

ESP_A2D_CONNECTION_STATE_CONNECTED
connection established

ESP_A2D_CONNECTION_STATE_DISCONNECTING
disconnecting remote device

enum **esp_a2d_disc_rsn_t**
Bluetooth A2DP disconnection reason.

Values:

ESP_A2D_DISC_RSN_NORMAL = 0
Finished disconnection that is initiated by local or remote device

ESP_A2D_DISC_RSN_ABNORMAL
Abnormal disconnection caused by signal loss

enum **esp_a2d_audio_state_t**
Bluetooth A2DP datapath states.

Values:

ESP_A2D_AUDIO_STATE_REMOTE_SUSPEND = 0
audio stream datapath suspended by remote device

ESP_A2D_AUDIO_STATE_STOPPED
audio stream datapath stopped

ESP_A2D_AUDIO_STATE_STARTED
audio stream datapath started

enum **esp_a2d_media_ctrl_ack_t**
A2DP media control command acknowledgement code.

Values:

ESP_A2D_MEDIA_CTRL_ACK_SUCCESS = 0
media control command is acknowledged with success

ESP_A2D_MEDIA_CTRL_ACK_FAILURE
media control command is acknowledged with failure

ESP_A2D_MEDIA_CTRL_ACK_BUSY
media control command is rejected, as previous command is not yet acknowledged

enum esp_a2d_media_ctrl_t
A2DP media control commands.

Values:

ESP_A2D_MEDIA_CTRL_NONE = 0
Not for application use, use inside stack only.

ESP_A2D_MEDIA_CTRL_CHECK_SRC_RDY
check whether AVDTP is connected, only used in A2DP source

ESP_A2D_MEDIA_CTRL_START
command to set up media transmission channel

ESP_A2D_MEDIA_CTRL_STOP
command to stop media transmission

ESP_A2D_MEDIA_CTRL_SUSPEND
command to suspend media transmission

enum esp_a2d_init_state_t
Bluetooth A2DP Initiation states.

Values:

ESP_A2D_DEINIT_SUCCESS = 0
A2DP profile deinit successful event

ESP_A2D_INIT_SUCCESS
A2DP profile deinit successful event

enum esp_a2d_cb_event_t
A2DP callback events.

Values:

ESP_A2D_CONNECTION_STATE_EVT = 0
connection state changed event

ESP_A2D_AUDIO_STATE_EVT
audio stream transmission state changed event

ESP_A2D_AUDIO_CFG_EVT
audio codec is configured, only used for A2DP SINK

ESP_A2D_MEDIA_CTRL_ACK_EVT
acknowledge event in response to media control commands

ESP_A2D_PROF_STATE_EVT
indicate a2dp init&deinit complete

BT AVRCP APIs

Overview Bluetooth AVRCP reference APIs.

[Instructions](#)

Application Example [Instructions](#)

API Reference

Header File

- `bt/host/bluetooth/api/include/api/esp_avrc_api.h`

Functions

esp_err_t **esp_avrc_ct_register_callback** (*esp_avrc_ct_cb_t* callback)

Register application callbacks to AVRCP module. This function should be called after `esp_bluetooth_enable()` completes successfully.

Return

- `ESP_OK`: success
- `ESP_ERR_INVALID_STATE`: if bluetooth stack is not yet enabled
- `ESP_FAIL`: others

Parameters

- [in] `callback`: AVRCP controller callback function

esp_err_t **esp_avrc_ct_init** (void)

Initialize the bluetooth AVRCP controller module. This function should be called after `esp_bluetooth_enable()` completes successfully. Note: AVRCP cannot work independently, AVRCP should be used along with A2DP and AVRCP should be initialized before A2DP.

Return

- `ESP_OK`: success
- `ESP_ERR_INVALID_STATE`: if bluetooth stack is not yet enabled
- `ESP_FAIL`: others

esp_err_t **esp_avrc_ct_deinit** (void)

De-initialize AVRCP controller module. This function should be called after `esp_bluetooth_enable()` completes successfully. Note: AVRCP cannot work independently, AVRCP should be used along with A2DP and AVRCP should be deinitialized before A2DP.

Return

- `ESP_OK`: success
- `ESP_ERR_INVALID_STATE`: if bluetooth stack is not yet enabled
- `ESP_FAIL`: others

esp_err_t **esp_avrc_ct_send_set_player_value_cmd** (uint8_t *tl*, uint8_t *attr_id*, uint8_t *value_id*)

Send player application settings command to AVRCP target. This function should be called after `ESP_AVRC_CT_CONNECTION_STATE_EVT` is received and AVRCP connection is established.

Return

- `ESP_OK`: success
- `ESP_ERR_INVALID_STATE`: if bluetooth stack is not yet enabled
- `ESP_FAIL`: others

Parameters

- [in] `tl`: transaction label, 0 to 15, consecutive commands should use different values.
- [in] `attr_id`: player application setting attribute IDs from one of `esp_avrc_ps_attr_ids_t`
- [in] `value_id`: attribute value defined for the specific player application setting attribute

esp_err_t **esp_avrc_ct_send_get_rn_capabilities_cmd** (uint8_t *tl*)

Send GetCapabilities PDU to AVRCP target to retrieve remote device's supported notification event_ids. This function should be called after `ESP_AVRC_CT_CONNECTION_STATE_EVT` is received and AVRCP connection is established.

Return

- `ESP_OK`: success
- `ESP_ERR_INVALID_STATE`: if bluetooth stack is not yet enabled
- `ESP_FAIL`: others

Parameters

- [in] `tl`: transaction label, 0 to 15, consecutive commands should use different values.

esp_err_t **esp_avrc_ct_send_register_notification_cmd**(uint8_t *tl*, uint8_t *event_id*,
uint32_t *event_parameter*)

Send register notification command to AVRCP target, This function should be called after ESP_AVRC_CT_CONNECTION_STATE_EVT is received and AVRCP connection is established.

Return

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_ERR_NOT_SUPPORTED: if the event_id is not supported in current implementation
- ESP_FAIL: others

Parameters

- [in] *tl*: : transaction label, 0 to 15, consecutive commands should use different values.
- [in] *event_id*: : id of events, e.g. ESP_AVRC_RN_PLAY_STATUS_CHANGE, ESP_AVRC_RN_TRACK_CHANGE, etc.
- [in] *event_parameter*: : playback interval for ESP_AVRC_RN_PLAY_POS_CHANGED; For other events , value of this parameter is ignored.

esp_err_t **esp_avrc_ct_send_set_absolute_volume_cmd**(uint8_t *tl*, uint8_t *volume*)

Send set absolute volume command to AVRCP target, This function should be called after ESP_AVRC_CT_CONNECTION_STATE_EVT is received and AVRCP connection is established.

Return

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_ERR_NOT_SUPPORTED: if the event_id is not supported in current implementation
- ESP_FAIL: others

Parameters

- [in] *tl*: : transaction label, 0 to 15, consecutive commands should use different values.
- [in] *volume*: : volume, 0 to 0x7f, means 0% to 100%

esp_err_t **esp_avrc_ct_send_metadata_cmd**(uint8_t *tl*, uint8_t *attr_mask*)

Send metadata command to AVRCP target, This function should be called after ESP_AVRC_CT_CONNECTION_STATE_EVT is received and AVRCP connection is established.

Return

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- [in] *tl*: : transaction label, 0 to 15, consecutive commands should use different values.
- [in] *attr_mask*: : mask of attributes, e.g. ESP_AVRC_MD_ATTR_ID_TITLE | ESP_AVRC_MD_ATTR_ID_ARTIST.

esp_err_t **esp_avrc_ct_send_passthrough_cmd**(uint8_t *tl*, uint8_t *key_code*, uint8_t *key_state*)

Send passthrough command to AVRCP target, This function should be called after ESP_AVRC_CT_CONNECTION_STATE_EVT is received and AVRCP connection is established.

Return

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- [in] *tl*: : transaction label, 0 to 15, consecutive commands should use different values.
- [in] *key_code*: : passthrough command code, e.g. ESP_AVRC_PT_CMD_PLAY, ESP_AVRC_PT_CMD_STOP, etc.
- [in] *key_state*: : passthrough command key state, ESP_AVRC_PT_CMD_STATE_PRESSED or ESP_AVRC_PT_CMD_STATE_RELEASED

esp_err_t **esp_avrc_tg_register_callback**(*esp_avrc_tg_cb_t* *callback*)

Register application callbacks to AVRCP target module; This function should be called after esp_bluedroid_enable() completes successfully.

Return

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- [in] *callback*: AVRCP target callback function

esp_err_t **esp_avrc_tg_init** (void)

Initialize the bluetooth AVRCP target module. This function should be called after `esp_bluedroid_enable()` completes successfully. Note: AVRCP cannot work independently, AVRCP should be used along with A2DP and AVRCP should be initialized before A2DP.

Return

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_avrc_tg_deinit** (void)

De-initialize AVRCP target module. This function should be called after `esp_bluedroid_enable()` completes successfully. Note: AVRCP cannot work independently, AVRCP should be used along with A2DP and AVRCP should be deinitialized before A2DP.

Return

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_avrc_tg_get_psth_cmd_filter** (*esp_avrc_psth_filter_t* *filter*,
esp_avrc_psth_bit_mask_t **cmd_set*)

Get the current filter of remote passthrough commands on AVRCP target. Filter is given by filter type and bit mask for the passthrough commands. This function should be called after `esp_avrc_tg_init()`. For filter type `ESP_AVRC_PSTH_FILTER_ALLOWED_CMD`, the retrieved command set is constant and it covers all of the passthrough commands that can possibly be supported. For filter type `ESP_AVRC_PSTH_FILTER_SUPPORT_COMMANDS`, the retrieved command set covers the passthrough commands selected to be supported according to current configuration. The configuration can be changed using `esp_avrc_tg_set_psth_cmd_filter()`

Return

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not enabled or AVRCP TG is not initialized
- ESP_ERR_INVALID_ARG: if filter type is invalid or *cmd_set* is NULL
- ESP_FAIL: otherwise

esp_err_t **esp_avrc_tg_set_psth_cmd_filter** (*esp_avrc_psth_filter_t* *filter*, **const**
esp_avrc_psth_bit_mask_t **cmd_set*)

Set the filter of remote passthrough commands on AVRCP target. Filter is given by filter type and bit mask for the passthrough commands. This function should be called after `esp_avrc_tg_init()`. If filter type is `ESP_AVRC_PSTH_FILTER_SUPPORT_CMD`, the passthrough commands which are set “1” as given in *cmd_set* will generate `ESP_AVRC_CT_PASSTHROUGH_RSP_EVT` callback event and are auto-accepted in the protocol stack, other commands are replied with response type “NOT IMPLEMENTED” (8). The set of supported commands should be a subset of allowed command set. The allowed command set can be retrieved using `esp_avrc_tg_get_psth_cmd_filter()` with filter type “`ESP_AVRC_PSTH_FILTER_ALLOWED_CMD`”

Filter type “`ESP_AVRC_PSTH_FILTER_ALLOWED_CMD`” does not apply to this function

Return

- ESP_OK: success
- ESP_ERR_INVALID_STATE: if bluetooth stack is not enabled
- ESP_ERR_INVALID_ARG: if filter type is invalid or *cmd_set* is NULL
- ESP_ERR_NOT_SUPPORTED: if filter type is `ESP_AVRC_PSTH_FILTER_ALLOWED_CMD`, or *cmd_set* includes unallowed commands

```
bool esp_avrc_psth_bit_mask_operation(esp_avrc_bit_mask_op_t op,
                                     esp_avrc_psth_bit_mask_t *psth, esp_avrc_pt_cmd_t cmd)
```

Operate on the type `esp_avrc_psth_bit_mask_t` with regard to a specific PASSTHROUGH command.

Return For operation `ESP_AVRC_BIT_MASK_OP_SET` or `ESP_AVRC_BIT_MASK_OP_CLEAR`, return true for a successful operation, otherwise return false. For operation `ESP_AVRC_BIT_MASK_OP_TEST`, return true if the corresponding bit is set, otherwise false.

Parameters

- [in] `op`: operation requested on the bit mask field
- [in] `psth`: pointer to passthrough command bit mask structure
- [in] `cmd`: passthrough command code

```
esp_err_t esp_avrc_tg_get_rn_evt_cap(esp_avrc_rn_evt_cap_t cap, esp_avrc_rn_evt_cap_mask_t *evt_set)
```

Get the requested event notification capabilities on local AVRC target. The capability is returned in a bit mask representation in `evt_set`. This function should be called after `esp_avrc_tg_init()`. For capability type `ESP_AVRC_RN_CAP_ALLOWED_EVT`, the retrieved event set is constant and it covers all of the notification events that can possibly be supported with current implementation. For capability type `ESP_AVRC_RN_CAP_SUPPORTED_EVT`, the event set covers the notification events selected to be supported under current configuration. The configuration can be changed using `esp_avrc_tg_set_rn_evt_cap()`.

Return

- `ESP_OK`: success
- `ESP_ERR_INVALID_STATE`: if bluetooth stack is not enabled or AVRC TG is not initialized
- `ESP_ERR_INVALID_ARG`: if `cap` is invalid or `evt_set` is NULL
- `ESP_FAIL`: otherwise

```
esp_err_t esp_avrc_tg_set_rn_evt_cap(const esp_avrc_rn_evt_cap_mask_t *evt_set)
```

Set the event notification capabilities on local AVRCP target. The capability is given in a bit mask representation in `evt_set` and must be a subset of allowed event IDs with current implementation. This function should be called after `esp_avrc_tg_init()`.

Return

- `ESP_OK`: success
- `ESP_ERR_INVALID_STATE`: if bluetooth stack is not enabled
- `ESP_ERR_INVALID_ARG`: if `evt_set` is NULL

```
bool esp_avrc_rn_evt_bit_mask_operation(esp_avrc_bit_mask_op_t op,
                                       esp_avrc_rn_evt_cap_mask_t *events,
                                       esp_avrc_rn_event_ids_t event_id)
```

Operate on the type `esp_avrc_rn_evt_cap_mask_t` with regard to a specific event.

Return For operation `ESP_AVRC_BIT_MASK_OP_SET` or `ESP_AVRC_BIT_MASK_OP_CLEAR`, return true for a successful operation, otherwise return false. For operation `ESP_AVRC_BIT_MASK_OP_TEST`, return true if the corresponding bit is set, otherwise false.

Parameters

- [in] `op`: operation requested on the bit mask field
- [in] `events`: pointer to event notification capability bit mask structure
- [in] `event_id`: notification event code

```
esp_err_t esp_avrc_tg_send_rn_rsp(esp_avrc_rn_event_ids_t event_id, esp_avrc_rn_rsp_t rsp,
                                  esp_avrc_rn_param_t *param)
```

Send RegisterNotification Response to remote AVRCP controller. Local event notification capability can be set using `esp_avrc_tg_set_rn_evt_cap()`, in a bit mask representation in `evt_set`. This function should be called after `esp_avrc_tg_init()`.

Return

- `ESP_OK`: success
- `ESP_ERR_INVALID_STATE`: if bluetooth stack is not enabled or AVRC TG is not initialized
- `ESP_ERR_INVALID_ARG`: if `evt_set` is NULL

Parameters

- [in] `event_id`: notification event ID that remote AVRCP CT registers

- [in] `rsp`: notification response code
- [in] `param`: parameters included in the specific notification

Unions

union `esp_avrc_rn_param_t`
#include <esp_avrc_api.h> AVRCP notification parameters.

Public Members

`uint8_t` **volume**
response data for `ESP_AVRC_RN_VOLUME_CHANGE`, ranges 0..127

`esp_avrc_playback_stat_t` **playback**
response data for `ESP_AVRC_RN_PLAY_STATUS_CHANGE`

`uint8_t` **elm_id**[8]
response data for `ESP_AVRC_RN_TRACK_CHANGE`

`uint32_t` **play_pos**
response data for `ESP_AVRC_RN_PLAY_POS_CHANGED`, in millisecond

`esp_avrc_batt_stat_t` **batt**
response data for `ESP_AVRC_RN_BATTERY_STATUS_CHANGE`

union `esp_avrc_ct_cb_param_t`
#include <esp_avrc_api.h> AVRCP controller callback parameters.

Public Members

struct `esp_avrc_ct_cb_param_t::avrc_ct_conn_stat_param` **conn_stat**
AVRCP connection status

struct `esp_avrc_ct_cb_param_t::avrc_ct_psth_rsp_param` **psth_rsp**
passthrough command response

struct `esp_avrc_ct_cb_param_t::avrc_ct_meta_rsp_param` **meta_rsp**
metadata attributes response

struct `esp_avrc_ct_cb_param_t::avrc_ct_change_notify_param` **change_ntf**
notifications

struct `esp_avrc_ct_cb_param_t::avrc_ct_rmt_feats_param` **rmt_feats**
AVRCP features discovered from remote SDP server

struct `esp_avrc_ct_cb_param_t::avrc_ct_get_rn_caps_rsp_param` **get_rn_caps_rsp**
get supported event capabilities response from AVRCP target

struct `esp_avrc_ct_cb_param_t::avrc_ct_set_volume_rsp_param` **set_volume_rsp**
set absolute volume response event

struct `avrc_ct_change_notify_param`
#include <esp_avrc_api.h> `ESP_AVRC_CT_CHANGE_NOTIFY_EVT`.

Public Members

`uint8_t` **event_id**
id of AVRCP event notification

`esp_avrc_rn_param_t` **event_parameter**
event notification parameter

```
struct avrc_ct_conn_stat_param  
#include <esp_avrc_api.h> ESP_AVRC_CT_CONNECTION_STATE_EVT.
```

Public Members

bool **connected**
whether AVRC connection is set up

esp_bd_addr_t **remote_bda**
remote bluetooth device address

```
struct avrc_ct_get_rn_caps_rsp_param  
#include <esp_avrc_api.h> ESP_AVRC_CT_GET_RN_CAPABILITIES_RSP_EVT.
```

Public Members

uint8_t **cap_count**
number of items provided in event or company_id according to cap_id used

esp_avrc_rn_evt_cap_mask_t **evt_set**
supported event_ids represented in bit-mask

```
struct avrc_ct_meta_rsp_param  
#include <esp_avrc_api.h> ESP_AVRC_CT_METADATA_RSP_EVT.
```

Public Members

uint8_t **attr_id**
id of metadata attribute

uint8_t ***attr_text**
attribute itself

int **attr_length**
attribute character length

```
struct avrc_ct_psth_rsp_param  
#include <esp_avrc_api.h> ESP_AVRC_CT_PASSTHROUGH_RSP_EVT.
```

Public Members

uint8_t **t1**
transaction label, 0 to 15

uint8_t **key_code**
passthrough command code

uint8_t **key_state**
0 for PRESSED, 1 for RELEASED

```
struct avrc_ct_rmt_feats_param  
#include <esp_avrc_api.h> ESP_AVRC_CT_REMOTE_FEATURES_EVT.
```

Public Members

uint32_t **feat_mask**
AVRC feature mask of remote device

`uint16_t tg_feat_flag`
feature flag of remote device as TG

`esp_bd_addr_t remote_bda`
remote bluetooth device address

`struct avrc_ct_set_volume_rsp_param`
`#include <esp_avrc_api.h> ESP_AVRC_CT_SET_ABSOLUTE_VOLUME_RSP_EVT.`

Public Members

`uint8_t volume`
the volume which has actually been set, range is 0 to 0x7f, means 0% to 100%

`union esp_avrc_tg_cb_param_t`
`#include <esp_avrc_api.h> AVRC target callback parameters.`

Public Members

`struct esp_avrc_tg_cb_param_t::avrc_tg_conn_stat_param conn_stat`
AVRC connection status

`struct esp_avrc_tg_cb_param_t::avrc_tg_rmt_feats_param rmt_feats`
AVRC features discovered through SDP

`struct esp_avrc_tg_cb_param_t::avrc_tg_psth_cmd_param psth_cmd`
passthrough command

`struct esp_avrc_tg_cb_param_t::avrc_tg_set_abs_vol_param set_abs_vol`
set absolute volume command targeted on audio sink

`struct esp_avrc_tg_cb_param_t::avrc_tg_reg_ntf_param reg_ntf`
register notification

`struct esp_avrc_tg_cb_param_t::avrc_tg_set_app_value_param set_app_value`
set player application value

`struct avrc_tg_conn_stat_param`
`#include <esp_avrc_api.h> ESP_AVRC_TG_CONNECTION_STATE_EVT.`

Public Members

`bool connected`
whether AVRC connection is set up

`esp_bd_addr_t remote_bda`
remote bluetooth device address

`struct avrc_tg_psth_cmd_param`
`#include <esp_avrc_api.h> ESP_AVRC_TG_PASSTHROUGH_CMD_EVT.`

Public Members

`uint8_t key_code`
passthrough command code

`uint8_t key_state`
0 for PRESSED, 1 for RELEASED

`struct avrc_tg_reg_ntf_param`
`#include <esp_avrc_api.h> ESP_AVRC_TG_REGISTER_NOTIFICATION_EVT.`

Public Members

uint8_t event_id
event id of AVRC RegisterNotification

uint32_t event_parameter
event notification parameter

struct avrc_tg_rmt_feats_param
#include <esp_avrc_api.h> ESP_AVRC_TG_REMOTE_FEATURES_EVT.

Public Members

uint32_t feat_mask
AVRC feature mask of remote device

uint16_t ct_feat_flag
feature flag of remote device as CT

esp_bd_addr_t **remote_bda**
remote bluetooth device address

struct avrc_tg_set_abs_vol_param
#include <esp_avrc_api.h> ESP_AVRC_TG_SET_ABSOLUTE_VOLUME_CMD_EVT.

Public Members

uint8_t volume
volume ranges from 0 to 127

struct avrc_tg_set_app_value_param
#include <esp_avrc_api.h> ESP_AVRC_TG_SET_PLAYER_APP_VALUE_EVT.

Public Members

uint8_t num_val
attribute num

esp_avrc_set_app_value_param_t ***p_vals**
point to the id and value of player application attribute

Structures

struct esp_avrc_psth_bit_mask_t
AVRC passthrough command bit mask.

Public Members

uint16_t bits[8]
bit mask representation of PASSTHROUGH commands

struct esp_avrc_rn_evt_cap_mask_t
AVRC target notification event capability bit mask.

Public Members

uint16_t bits
bit mask representation of PASSTHROUGH commands

struct esp_avrc_set_app_value_param_t
AVRCP set app value parameters.

Public Members

uint8_t attr_id
player application attribute id

uint8_t attr_val
player application attribute value

Macros

ESP_AVRC_TRANS_LABEL_MAX
max transaction label

Type Definitions

typedef void (*esp_avrc_ct_cb_t) (esp_avrc_ct_cb_event_t event, esp_avrc_ct_cb_param_t *param)
AVRCP controller callback function type.

Parameters

- event: : Event type
- param: : Pointer to callback parameter union

typedef void (*esp_avrc_tg_cb_t) (esp_avrc_tg_cb_event_t event, esp_avrc_tg_cb_param_t *param)
AVRCP target callback function type.

Parameters

- event: : Event type
- param: : Pointer to callback parameter union

Enumerations

enum esp_avrc_features_t
AVRC feature bit mask.

Values:

ESP_AVRC_FEAT_RCTG = 0x0001
remote control target

ESP_AVRC_FEAT_RCCT = 0x0002
remote control controller

ESP_AVRC_FEAT_VENDOR = 0x0008
remote control vendor dependent commands

ESP_AVRC_FEAT_BROWSE = 0x0010
use browsing channel

ESP_AVRC_FEAT_META_DATA = 0x0040
remote control metadata transfer command/response

ESP_AVRC_FEAT_ADV_CTRL = 0x0200
remote control advanced control commmand/response

enum esp_avrc_feature_flag_t

AVRC supported features flag retrieved in SDP record.

*Values:***ESP_AVRC_FEAT_FLAG_CAT1** = 0x0001
category 1**ESP_AVRC_FEAT_FLAG_CAT2** = 0x0002
category 2**ESP_AVRC_FEAT_FLAG_CAT3** = 0x0004
category 3**ESP_AVRC_FEAT_FLAG_CAT4** = 0x0008
category 4**ESP_AVRC_FEAT_FLAG_BROWSING** = 0x0040
browsing**ESP_AVRC_FEAT_FLAG_COVER_ART_GET_IMAGE_PROP** = 0x0080
Cover Art GetImageProperties**ESP_AVRC_FEAT_FLAG_COVER_ART_GET_IMAGE** = 0x0100
Cover Art GetImage**ESP_AVRC_FEAT_FLAG_COVER_ART_GET_LINKED_THUMBNAIL** = 0x0200
Cover Art GetLinkedThumbnail**enum esp_avrc_pt_cmd_t**

AVRC passthrough command code.

*Values:***ESP_AVRC_PT_CMD_SELECT** = 0x00
select**ESP_AVRC_PT_CMD_UP** = 0x01
up**ESP_AVRC_PT_CMD_DOWN** = 0x02
down**ESP_AVRC_PT_CMD_LEFT** = 0x03
left**ESP_AVRC_PT_CMD_RIGHT** = 0x04
right**ESP_AVRC_PT_CMD_RIGHT_UP** = 0x05
right-up**ESP_AVRC_PT_CMD_RIGHT_DOWN** = 0x06
right-down**ESP_AVRC_PT_CMD_LEFT_UP** = 0x07
left-up**ESP_AVRC_PT_CMD_LEFT_DOWN** = 0x08
left-down**ESP_AVRC_PT_CMD_ROOT_MENU** = 0x09
root menu**ESP_AVRC_PT_CMD_SETUP_MENU** = 0x0A
setup menu**ESP_AVRC_PT_CMD_CONT_MENU** = 0x0B
contents menu

ESP_AVRC_PT_CMD_FAV_MENU = 0x0C
favorite menu

ESP_AVRC_PT_CMD_EXIT = 0x0D
exit

ESP_AVRC_PT_CMD_0 = 0x20
0

ESP_AVRC_PT_CMD_1 = 0x21
1

ESP_AVRC_PT_CMD_2 = 0x22
2

ESP_AVRC_PT_CMD_3 = 0x23
3

ESP_AVRC_PT_CMD_4 = 0x24
4

ESP_AVRC_PT_CMD_5 = 0x25
5

ESP_AVRC_PT_CMD_6 = 0x26
6

ESP_AVRC_PT_CMD_7 = 0x27
7

ESP_AVRC_PT_CMD_8 = 0x28
8

ESP_AVRC_PT_CMD_9 = 0x29
9

ESP_AVRC_PT_CMD_DOT = 0x2A
dot

ESP_AVRC_PT_CMD_ENTER = 0x2B
enter

ESP_AVRC_PT_CMD_CLEAR = 0x2C
clear

ESP_AVRC_PT_CMD_CHAN_UP = 0x30
channel up

ESP_AVRC_PT_CMD_CHAN_DOWN = 0x31
channel down

ESP_AVRC_PT_CMD_PREV_CHAN = 0x32
previous channel

ESP_AVRC_PT_CMD_SOUND_SEL = 0x33
sound select

ESP_AVRC_PT_CMD_INPUT_SEL = 0x34
input select

ESP_AVRC_PT_CMD_DISP_INFO = 0x35
display information

ESP_AVRC_PT_CMD_HELP = 0x36
help

ESP_AVRC_PT_CMD_PAGE_UP = 0x37
page up

ESP_AVRC_PT_CMD_PAGE_DOWN = 0x38
page down

ESP_AVRC_PT_CMD_POWER = 0x40
power

ESP_AVRC_PT_CMD_VOL_UP = 0x41
volume up

ESP_AVRC_PT_CMD_VOL_DOWN = 0x42
volume down

ESP_AVRC_PT_CMD_MUTE = 0x43
mute

ESP_AVRC_PT_CMD_PLAY = 0x44
play

ESP_AVRC_PT_CMD_STOP = 0x45
stop

ESP_AVRC_PT_CMD_PAUSE = 0x46
pause

ESP_AVRC_PT_CMD_RECORD = 0x47
record

ESP_AVRC_PT_CMD_REWIND = 0x48
rewind

ESP_AVRC_PT_CMD_FAST_FORWARD = 0x49
fast forward

ESP_AVRC_PT_CMD_EJECT = 0x4A
eject

ESP_AVRC_PT_CMD_FORWARD = 0x4B
forward

ESP_AVRC_PT_CMD_BACKWARD = 0x4C
backward

ESP_AVRC_PT_CMD_ANGLE = 0x50
angle

ESP_AVRC_PT_CMD_SUBPICT = 0x51
subpicture

ESP_AVRC_PT_CMD_F1 = 0x71
F1

ESP_AVRC_PT_CMD_F2 = 0x72
F2

ESP_AVRC_PT_CMD_F3 = 0x73
F3

ESP_AVRC_PT_CMD_F4 = 0x74
F4

ESP_AVRC_PT_CMD_F5 = 0x75
F5

ESP_AVRC_PT_CMD_VENDOR = 0x7E
vendor unique

enum esp_avrc_psth_filter_t
AVRC passthrough command filter.

Values:

ESP_AVRC_PSTH_FILTER_ALLOWED_CMD = 0
all of the PASSTHROUGH commands that can possibly be used, immutable

ESP_AVRC_PSTH_FILTER_SUPPORTED_CMD = 1
PASSTHROUGH commands selectively supported according to the current configuration

ESP_AVRC_PSTH_FILTER_SUPPORT_MAX

enum esp_avrc_bit_mask_op_t

Values:

ESP_AVRC_BIT_MASK_OP_TEST = 0
operation code to test a specific bit

ESP_AVRC_BIT_MASK_OP_SET = 1
operation code to set a specific bit

ESP_AVRC_BIT_MASK_OP_CLEAR = 2
operation code to clear a specific bit

enum esp_avrc_pt_cmd_state_t

AVRC passthrough command state.

Values:

ESP_AVRC_PT_CMD_STATE_PRESSED = 0
key pressed

ESP_AVRC_PT_CMD_STATE_RELEASED = 1
key released

enum esp_avrc_ct_cb_event_t

AVRC Controller callback events.

Values:

ESP_AVRC_CT_CONNECTION_STATE_EVT = 0
connection state changed event

ESP_AVRC_CT_PASSTHROUGH_RSP_EVT = 1
passthrough response event

ESP_AVRC_CT_METADATA_RSP_EVT = 2
metadata response event

ESP_AVRC_CT_PLAY_STATUS_RSP_EVT = 3
play status response event

ESP_AVRC_CT_CHANGE_NOTIFY_EVT = 4
notification event

ESP_AVRC_CT_REMOTE_FEATURES_EVT = 5
feature of remote device indication event

ESP_AVRC_CT_GET_RN_CAPABILITIES_RSP_EVT = 6
supported notification events capability of peer device

ESP_AVRC_CT_SET_ABSOLUTE_VOLUME_RSP_EVT = 7
set absolute volume response event

enum esp_avrc_tg_cb_event_t

AVRC Target callback events.

Values:

ESP_AVRC_TG_CONNECTION_STATE_EVT = 0
connection state changed event

ESP_AVRC_TG_REMOTE_FEATURES_EVT = 1
feature of remote device indication event

ESP_AVRC_TG_PASSTHROUGH_CMD_EVT = 2
passthrough command event

ESP_AVRC_TG_SET_ABSOLUTE_VOLUME_CMD_EVT = 3
set absolute volume command from remote device

ESP_AVRC_TG_REGISTER_NOTIFICATION_EVT = 4
register notification event

ESP_AVRC_TG_SET_PLAYER_APP_VALUE_EVT = 5
set application attribute value, attribute refer to esp_avrc_ps_attr_ids_t

enum esp_avrc_md_attr_mask_t
AVRC metadata attribute mask.

Values:

ESP_AVRC_MD_ATTR_TITLE = 0x1
title of the playing track

ESP_AVRC_MD_ATTR_ARTIST = 0x2
track artist

ESP_AVRC_MD_ATTR_ALBUM = 0x4
album name

ESP_AVRC_MD_ATTR_TRACK_NUM = 0x8
track position on the album

ESP_AVRC_MD_ATTR_NUM_TRACKS = 0x10
number of tracks on the album

ESP_AVRC_MD_ATTR_GENRE = 0x20
track genre

ESP_AVRC_MD_ATTR_PLAYING_TIME = 0x40
total album playing time in milliseconds

enum esp_avrc_rn_event_ids_t
AVRC event notification ids.

Values:

ESP_AVRC_RN_PLAY_STATUS_CHANGE = 0x01
track status change, eg. from playing to paused

ESP_AVRC_RN_TRACK_CHANGE = 0x02
new track is loaded

ESP_AVRC_RN_TRACK_REACHED_END = 0x03
current track reached end

ESP_AVRC_RN_TRACK_REACHED_START = 0x04
current track reached start position

ESP_AVRC_RN_PLAY_POS_CHANGED = 0x05
track playing position changed

ESP_AVRC_RN_BATTERY_STATUS_CHANGE = 0x06
battery status changed

ESP_AVRC_RN_SYSTEM_STATUS_CHANGE = 0x07
system status changed

ESP_AVRC_RN_APP_SETTING_CHANGE = 0x08
application settings changed

ESP_AVRC_RN_NOW_PLAYING_CHANGE = 0x09
now playing content changed

ESP_AVRC_RN_AVAILABLE_PLAYERS_CHANGE = 0x0a
available players changed

ESP_AVRC_RN_ADDRESSED_PLAYER_CHANGE = 0x0b
the addressed player changed

ESP_AVRC_RN_UIDS_CHANGE = 0x0c
UIDs changed

ESP_AVRC_RN_VOLUME_CHANGE = 0x0d
volume changed locally on TG

ESP_AVRC_RN_MAX_EVT

enum esp_avrc_rn_evt_cap_t
AVRC target notification event notification capability.

Values:

ESP_AVRC_RN_CAP_ALLOWED_EVT = 0
all of the notification events that can possibly be supported, immutable

ESP_AVRC_RN_CAP_SUPPORTED_EVT = 1
notification events selectively supported according to the current configuration

ESP_AVRC_RN_CAP_MAX

enum esp_avrc_rn_rsp_t
AVRC notification response type.

Values:

ESP_AVRC_RN_RSP_INTERIM = 13
initial response to RegisterNotification, should be sent T_mtp(1000ms) from receiving the command

ESP_AVRC_RN_RSP_CHANGED = 15
final response to RegisterNotification command

enum esp_avrc_ps_attr_ids_t
AVRC player setting ids.

Values:

ESP_AVRC_PS_EQUALIZER = 0x01
equalizer, on or off

ESP_AVRC_PS_REPEAT_MODE = 0x02
repeat mode

ESP_AVRC_PS_SHUFFLE_MODE = 0x03
shuffle mode

ESP_AVRC_PS_SCAN_MODE = 0x04
scan mode on or off

ESP_AVRC_PS_MAX_ATTR

enum esp_avrc_ps_eq_value_ids_t
AVRC equalizer modes.

Values:

ESP_AVRC_PS_EQUALIZER_OFF = 0x1
equalizer OFF

ESP_AVRC_PS_EQUALIZER_ON = 0x2
equalizer ON

enum esp_avrc_ps_rpt_value_ids_t
AVRC repeat modes.

Values:

ESP_AVRC_PS_REPEAT_OFF = 0x1
repeat mode off

ESP_AVRC_PS_REPEAT_SINGLE = 0x2
single track repeat

ESP_AVRC_PS_REPEAT_GROUP = 0x3
group repeat

enum esp_avrc_ps_shf_value_ids_t
AVRC shuffle modes.

Values:

ESP_AVRC_PS_SHUFFLE_OFF = 0x1

ESP_AVRC_PS_SHUFFLE_ALL = 0x2

ESP_AVRC_PS_SHUFFLE_GROUP = 0x3

enum esp_avrc_ps_scn_value_ids_t
AVRC scan modes.

Values:

ESP_AVRC_PS_SCAN_OFF = 0x1
scan off

ESP_AVRC_PS_SCAN_ALL = 0x2
all tracks scan

ESP_AVRC_PS_SCAN_GROUP = 0x3
group scan

enum esp_avrc_rsp_t
AVCTP response codes.

Values:

ESP_AVRC_RSP_NOT_IMPL = 8
not implemented

ESP_AVRC_RSP_ACCEPT = 9
accept

ESP_AVRC_RSP_REJECT = 10
reject

ESP_AVRC_RSP_IN_TRANS = 11
in transition

ESP_AVRC_RSP_IMPL_STBL = 12
implemented/stable

ESP_AVRC_RSP_CHANGED = 13
changed

ESP_AVRC_RSP_INTERIM = 15
interim

enum esp_avrc_batt_stat_t
AVRCP battery status.

Values:

ESP_AVRC_BATT_NORMAL = 0
normal state

ESP_AVRC_BATT_WARNING = 1
unable to operate soon

ESP_AVRC_BATT_CRITICAL = 2
cannot operate any more

ESP_AVRC_BATT_EXTERNAL = 3
plugged to external power supply

ESP_AVRC_BATT_FULL_CHARGE = 4
when completely charged from external power supply

enum esp_avrc_playback_stat_t
AVRCP current status of playback.

Values:

ESP_AVRC_PLAYBACK_STOPPED = 0
stopped

ESP_AVRC_PLAYBACK_PLAYING = 1
playing

ESP_AVRC_PLAYBACK_PAUSED = 2
paused

ESP_AVRC_PLAYBACK_FWD_SEEK = 3
forward seek

ESP_AVRC_PLAYBACK_REV_SEEK = 4
reverse seek

ESP_AVRC_PLAYBACK_ERROR = 0xFF
error

SPP API

Overview [Instructions](#)

Application Example Check `bluetooth/bluedroid/classic_bt` folder in ESP-IDF examples, which contains the following application:

- This is a SPP demo. This demo can discover the service, connect, send and receive SPP data `bluetooth/bluedroid/classic_bt/bt_spp_acceptor`, `bluetooth/bluedroid/classic_bt/bt_spp_initiator`

API Reference

Header File

- `bt/host/bluedroid/api/include/api/esp_spp_api.h`

Functions

`esp_err_t esp_spp_register_callback(esp_spp_cb_t callback)`

This function is called to init callbacks with SPP module.

Return

- `ESP_OK`: success
- other: failed

Parameters

- `[in] callback`: pointer to the init callback function.

`esp_err_t esp_spp_init(esp_spp_mode_t mode)`

This function is called to init SPP.

Return

- ESP_OK: success
- other: failed

Parameters

- [in] mode: Choose the mode of SPP, ESP_SPP_MODE_CB or ESP_SPP_MODE_VFS.

esp_err_t **esp_spp_deinit** (void)

This function is called to uninit SPP.

Return

- ESP_OK: success
- other: failed

esp_err_t **esp_spp_start_discovery** (*esp_bd_addr_t* bd_addr)

This function is called to performs service discovery for the services provided by the given peer device. When the operation is complete the callback function will be called with a ESP_SPP_DISCOVERY_COMP_EVT.

Return

- ESP_OK: success
- other: failed

Parameters

- [in] bd_addr: Remote device bluetooth device address.

esp_err_t **esp_spp_connect** (*esp_spp_sec_t* sec_mask, *esp_spp_role_t* role, uint8_t remote_scn, *esp_bd_addr_t* peer_bd_addr)

This function makes an SPP connection to a remote BD Address. When the connection is initiated or failed to initiate, the callback is called with ESP_SPP_CL_INIT_EVT. When the connection is established or failed, the callback is called with ESP_SPP_OPEN_EVT.

Return

- ESP_OK: success
- other: failed

Parameters

- [in] sec_mask: Security Setting Mask. Suggest to use ESP_SPP_SEC_NONE, ESP_SPP_SEC_AUTHORIZE or ESP_SPP_SEC_AUTHENTICATE only.
- [in] role: Master or slave.
- [in] remote_scn: Remote device bluetooth device SCN.
- [in] peer_bd_addr: Remote device bluetooth device address.

esp_err_t **esp_spp_disconnect** (uint32_t handle)

This function closes an SPP connection.

Return

- ESP_OK: success
- other: failed

Parameters

- [in] handle: The connection handle.

esp_err_t **esp_spp_start_srv** (*esp_spp_sec_t* sec_mask, *esp_spp_role_t* role, uint8_t local_scn, const char *name)

This function create a SPP server and starts listening for an SPP connection request from a remote Bluetooth device. When the server is started successfully, the callback is called with ESP_SPP_START_EVT. When the connection is established, the callback is called with ESP_SPP_SRV_OPEN_EVT.

Return

- ESP_OK: success
- other: failed

Parameters

- [in] sec_mask: Security Setting Mask. Suggest to use ESP_SPP_SEC_NONE, ESP_SPP_SEC_AUTHORIZE or ESP_SPP_SEC_AUTHENTICATE only.
- [in] role: Master or slave.
- [in] local_scn: The specific channel you want to get. If channel is 0, means get any channel.
- [in] name: Server' s name.

***esp_err_t* esp_spp_stop_srv** (void)

This function stops all SPP servers. The operation will close all active SPP connection first, then the callback function will be called with ESP_SPP_CLOSE_EVT, and the number of ESP_SPP_CLOSE_EVT is equal to the number of connection. When the operation is completed, the callback is called with ESP_SPP_SRV_STOP_EVT. This function must be called after esp_spp_init() successful and before esp_spp_deinit().

Return

- ESP_OK: success
- other: failed

***esp_err_t* esp_spp_stop_srv_scn** (uint8_t *scn*)

This function stops a specific SPP server. The operation will close all active SPP connection first on the specific SPP server, then the callback function will be called with ESP_SPP_CLOSE_EVT, and the number of ESP_SPP_CLOSE_EVT is equal to the number of connection. When the operation is completed, the callback is called with ESP_SPP_SRV_STOP_EVT. This function must be called after esp_spp_init() successful and before esp_spp_deinit().

Return

- ESP_OK: success
- other: failed

Parameters

- [in] *scn*: Server channel number.

***esp_err_t* esp_spp_write** (uint32_t *handle*, int *len*, uint8_t **p_data*)

This function is used to write data, only for ESP_SPP_MODE_CB.

Return

- ESP_OK: success
- other: failed

Parameters

- [in] *handle*: The connection handle.
- [in] *len*: The length of the data written.
- [in] *p_data*: The data written.

***esp_err_t* esp_spp_vfs_register** (void)

This function is used to register VFS.

Return

- ESP_OK: success
- other: failed

Unions**union esp_spp_cb_param_t**

#include <esp_spp_api.h> SPP callback parameters union.

Public Members

struct esp_spp_cb_param_t::spp_init_evt_param **init**
SPP callback param of SPP_INIT_EVT

struct esp_spp_cb_param_t::spp_uninit_evt_param **uninit**
SPP callback param of SPP_UNINIT_EVT

struct esp_spp_cb_param_t::spp_discovery_comp_evt_param **disc_comp**
SPP callback param of SPP_DISCOVERY_COMP_EVT

struct esp_spp_cb_param_t::spp_open_evt_param **open**
SPP callback param of ESP_SPP_OPEN_EVT

struct esp_spp_cb_param_t::spp_srv_open_evt_param **srv_open**
SPP callback param of ESP_SPP_SRV_OPEN_EVT

```
struct esp_spp_cb_param_t::spp_close_evt_param close  
    SPP callback param of ESP_SPP_CLOSE_EVT  
  
struct esp_spp_cb_param_t::spp_start_evt_param start  
    SPP callback param of ESP_SPP_START_EVT  
  
struct esp_spp_cb_param_t::spp_srv_stop_evt_param srv_stop  
    SPP callback param of ESP_SPP_SRV_STOP_EVT  
  
struct esp_spp_cb_param_t::spp_cl_init_evt_param cl_init  
    SPP callback param of ESP_SPP_CL_INIT_EVT  
  
struct esp_spp_cb_param_t::spp_write_evt_param write  
    SPP callback param of ESP_SPP_WRITE_EVT  
  
struct esp_spp_cb_param_t::spp_data_ind_evt_param data_ind  
    SPP callback param of ESP_SPP_DATA_IND_EVT  
  
struct esp_spp_cb_param_t::spp_cong_evt_param cong  
    SPP callback param of ESP_SPP_CONG_EVT  
  
struct spp_cl_init_evt_param  
    #include <esp_spp_api.h> ESP_SPP_CL_INIT_EVT.
```

Public Members

```
esp_spp_status_t status  
    status  
  
uint32_t handle  
    The connection handle  
  
uint8_t sec_id  
    security ID used by this server  
  
bool use_co  
    TRUE to use co_rfc_data  
  
struct spp_close_evt_param  
    #include <esp_spp_api.h> ESP_SPP_CLOSE_EVT.
```

Public Members

```
esp_spp_status_t status  
    status  
  
uint32_t port_status  
    PORT status  
  
uint32_t handle  
    The connection handle  
  
bool async  
    FALSE, if local initiates disconnect  
  
struct spp_cong_evt_param  
    #include <esp_spp_api.h> ESP_SPP_CONG_EVT.
```

Public Members

```
esp_spp_status_t status  
    status
```

uint32_t handle
The connection handle

bool cong
TRUE, congested. FALSE, uncongested

struct spp_data_ind_evt_param
#include <esp_spp_api.h> ESP_SPP_DATA_IND_EVT.

Public Members

esp_spp_status_t **status**
status

uint32_t handle
The connection handle

uint16_t len
The length of data

uint8_t *data
The data received

struct spp_discovery_comp_evt_param
#include <esp_spp_api.h> SPP_DISCOVERY_COMP_EVT.

Public Members

esp_spp_status_t **status**
status

uint8_t scn_num
The num of scn_num

uint8_t scn[ESP_SPP_MAX_SCN]
channel #

const char *service_name[ESP_SPP_MAX_SCN]
service_name

struct spp_init_evt_param
#include <esp_spp_api.h> SPP_INIT_EVT.

Public Members

esp_spp_status_t **status**
status

struct spp_open_evt_param
#include <esp_spp_api.h> ESP_SPP_OPEN_EVT.

Public Members

esp_spp_status_t **status**
status

uint32_t handle
The connection handle

int fd
The file descriptor only for ESP_SPP_MODE_VFS

esp_bd_addr_t **rem_bda**

The peer address

```
struct spp_srv_open_evt_param  
#include <esp_spp_api.h> ESP_SPP_SRV_OPEN_EVT.
```

Public Members

esp_spp_status_t **status**

status

uint32_t **handle**

The connection handle

uint32_t **new_listen_handle**

The new listen handle

int **fd**

The file descriptor only for ESP_SPP_MODE_VFS

esp_bd_addr_t **rem_bda**

The peer address

```
struct spp_srv_stop_evt_param  
#include <esp_spp_api.h> ESP_SPP_SRV_STOP_EVT.
```

Public Members

esp_spp_status_t **status**

status

uint8_t **scn**

Server channel number

```
struct spp_start_evt_param  
#include <esp_spp_api.h> ESP_SPP_START_EVT.
```

Public Members

esp_spp_status_t **status**

status

uint32_t **handle**

The connection handle

uint8_t **sec_id**

security ID used by this server

uint8_t **scn**

Server channel number

bool **use_co**

TRUE to use co_rfc_data

```
struct spp_uninit_evt_param  
#include <esp_spp_api.h> SPP_UNINIT_EVT.
```


Public Members

esp_spp_status_t **status**
status

struct spp_write_evt_param
#include <esp_spp_api.h> ESP_SPP_WRITE_EVT.

Public Members

esp_spp_status_t **status**
status

uint32_t handle
The connection handle

int len
The length of the data written.

bool cong
congestion status

Macros

ESP_SPP_SEC_NONE
No security. relate to BTA_SEC_NONE in bta/bta_api.h

ESP_SPP_SEC_AUTHORIZE
Authorization required (only needed for out going connection) relate to BTA_SEC_AUTHORIZE in bta/bta_api.h

ESP_SPP_SEC_AUTHENTICATE
Authentication required. relate to BTA_SEC_AUTHENTICATE in bta/bta_api.h

ESP_SPP_SEC_ENCRYPT
Encryption required. relate to BTA_SEC_ENCRYPT in bta/bta_api.h

ESP_SPP_SEC_MODE4_LEVEL4
Mode 4 level 4 service, i.e. incoming/outgoing MITM and P-256 encryption relate to BTA_SEC_MODE4_LEVEL4 in bta/bta_api.h

ESP_SPP_SEC_MITM
Man-In-The-Middle protection relate to BTA_SEC_MITM in bta/bta_api.h

ESP_SPP_SEC_IN_16_DIGITS
Min 16 digit for pin code relate to BTA_SEC_IN_16_DIGITS in bta/bta_api.h

ESP_SPP_MAX_MTU
SPP max MTU

ESP_SPP_MAX_SCN
SPP max SCN

Type Definitions

typedef uint16_t esp_spp_sec_t

typedef void (esp_spp_cb_t) (esp_spp_cb_event_t event, esp_spp_cb_param_t *param)
SPP callback function type.

Parameters

- event: Event type
- param: Point to callback parameter, currently is union type

Enumerations**enum esp_spp_status_t***Values:*

- ESP_SPP_SUCCESS** = 0
Successful operation.
- ESP_SPP_FAILURE**
Generic failure.
- ESP_SPP_BUSY**
Temporarily can not handle this request.
- ESP_SPP_NO_DATA**
No data
- ESP_SPP_NO_RESOURCE**
No more resource
- ESP_SPP_NEED_INIT**
SPP module shall init first
- ESP_SPP_NEED_DEINIT**
SPP module shall deinit first
- ESP_SPP_NO_CONNECTION**
Connection may have been closed
- ESP_SPP_NO_SERVER**
No SPP server

enum esp_spp_role_t*Values:*

- ESP_SPP_ROLE_MASTER** = 0
Role: master
- ESP_SPP_ROLE_SLAVE** = 1
Role: slave

enum esp_spp_mode_t*Values:*

- ESP_SPP_MODE_CB** = 0
When data is coming, a callback will come with data
- ESP_SPP_MODE_VFS** = 1
Use VFS to write/read data

enum esp_spp_cb_event_t

SPP callback function events.

Values:

- ESP_SPP_INIT_EVT** = 0
When SPP is initied, the event comes
- ESP_SPP_UNINIT_EVT** = 1
When SPP is uninitied, the event comes
- ESP_SPP_DISCOVERY_COMP_EVT** = 8
When SDP discovery complete, the event comes
- ESP_SPP_OPEN_EVT** = 26
When SPP Client connection open, the event comes
- ESP_SPP_CLOSE_EVT** = 27
When SPP connection closed, the event comes

ESP_SPP_START_EVT = 28

When SPP server started, the event comes

ESP_SPP_CL_INIT_EVT = 29

When SPP client initiated a connection, the event comes

ESP_SPP_DATA_IND_EVT = 30

When SPP connection received data, the event comes, only for ESP_SPP_MODE_CB

ESP_SPP_CONG_EVT = 31

When SPP connection congestion status changed, the event comes, only for ESP_SPP_MODE_CB

ESP_SPP_WRITE_EVT = 33

When SPP write operation completes, the event comes, only for ESP_SPP_MODE_CB

ESP_SPP_SRV_OPEN_EVT = 34

When SPP Server connection open, the event comes

ESP_SPP_SRV_STOP_EVT = 35

When SPP server stopped, the event comes

HFP DEFINES

[Overview](#) [Instructions](#)

API Reference

Header File

- [bt/host/bluetooth/api/include/api/esp_hf_defs.h](#)

Macros

ESP_BT_HF_NUMBER_LEN

ESP_BT_HF_OPERATOR_NAME_LEN

BTC_HSAG_SERVICE_NAME

BTC_HFAG_SERVICE_NAME

BTC_HF_SERVICES

BTC_HF_SERVICE_NAMES

BTC_HF_SECURITY

BTC_HF_CALL_END_TIMEOUT

BTC_HF_INVALID_IDX

Type Definitions

typedef void (*esp_hf_connection_state_callback) (esp_hf_connection_state_t state, esp_bd_addr_t *bd_addr)

Callback for connection state change. state will have one of the values from BtHfConnectionState

typedef void (*esp_hf_audio_state_callback) (esp_hf_audio_state_t state, esp_bd_addr_t *bd_addr)

Callback for audio connection state change. state will have one of the values from BtHfAudioState

typedef void (*esp_hf_vr_cmd_callback) (esp_hf_vr_state_t state, esp_bd_addr_t *bd_addr)

Callback for VR connection state change. state will have one of the values from BtHfVRState

typedef void (*esp_hf_answer_call_cmd_callback) (esp_bd_addr_t *bd_addr)

Callback for answer incoming call (ATA)

```

typedef void (*esp_hf_hangup_call_cmd_callback) (esp_bd_addr_t *bd_addr)
    Callback for disconnect call (AT+CHUP)

typedef void (*esp_hf_volume_cmd_callback) (esp_hf_volume_control_target_t type, int volume, esp_bd_addr_t *bd_addr)
    Callback for disconnect call (AT+CHUP) type will denote Speaker/Mic gain (BtHfVolumeControl).

typedef void (*esp_hf_dial_call_cmd_callback) (char *number, esp_bd_addr_t *bd_addr)
    Callback for dialing an outgoing call If number is NULL, redial

typedef void (*esp_hf_dtmf_cmd_callback) (char tone, esp_bd_addr_t *bd_addr)
    Callback for sending DTMF tones tone contains the dtmf character to be sent

typedef void (*esp_hf_nrec_cmd_callback) (esp_hf_nrec_t nrec, esp_bd_addr_t *bd_addr)
    Callback for enabling/disabling noise reduction/echo cancellation value will be 1 to enable, 0 to disable

typedef void (*esp_hf_wbs_callback) (esp_hf_wbs_config_t wbs, esp_bd_addr_t *bd_addr)
    Callback for AT+BCS and event from BAC WBS enable, WBS disable

typedef void (*esp_hf_chld_cmd_callback) (esp_hf_chld_type_t chld, esp_bd_addr_t *bd_addr)
    Callback for call hold handling (AT+CHLD) value will contain the call hold command (0, 1, 2, 3)

typedef void (*esp_hf_cnum_cmd_callback) (esp_bd_addr_t *bd_addr)
    Callback for CNUM (subscriber number)

typedef void (*esp_hf_cind_cmd_callback) (esp_bd_addr_t *bd_addr)
    Callback for indicators (CIND)

typedef void (*esp_hf_cops_cmd_callback) (esp_bd_addr_t *bd_addr)
    Callback for operator selection (COPS)

typedef void (*esp_hf_clcc_cmd_callback) (esp_bd_addr_t *bd_addr)
    Callback for call list (AT+CLCC)

typedef void (*esp_hf_unknown_at_cmd_callback) (char *at_string, esp_bd_addr_t *bd_addr)
    Callback for unknown AT command recd from AG at_string will contain the unparsed AT string

typedef void (*esp_hf_key_pressed_cmd_callback) (esp_bd_addr_t *bd_addr)
    Callback for keypressed (HSP) event.

```

Enumerations

```

enum esp_hf_in_band_ring_state_t
    in-band ring tone state

```

Values:

```

ESP_HF_IN_BAND_RINGTONE_NOT_PROVIDED = 0

```

```

ESP_HF_IN_BAND_RINGTONE_PROVIDED

```

```

enum esp_hf_vr_state_t
    voice recognition state

```

Values:

```

ESP_HF_VR_STATE_DISABLED = 0
    voice recognition disabled

```

```

ESP_HF_VR_STATE_ENABLED
    voice recognition enabled

```

```

enum esp_hf_volume_control_target_t
    Bluetooth HFP audio volume control target.

```

Values:

```

ESP_HF_VOLUME_CONTROL_TARGET_SPK = 0
    speaker

```

ESP_HF_VOLUME_CONTROL_TARGET_MIC
microphone

enum esp_hf_audio_state_t
Bluetooth HFP audio connection status.

Values:

ESP_HF_AUDIO_STATE_DISCONNECTED = 0
audio connection released

ESP_HF_AUDIO_STATE_CONNECTING
audio connection has been initiated

ESP_HF_AUDIO_STATE_CONNECTED
audio connection is established

ESP_HF_AUDIO_STATE_CONNECTED_MSBC
mSBC audio connection is established

enum esp_hf_volume_type_t

Values:

ESP_HF_VOLUME_TYPE_SPK = 0

ESP_HF_VOLUME_TYPE_MIC

enum esp_hf_network_state_t
+CIND network service availability status

Values:

ESP_HF_NETWORK_STATE_NOT_AVAILABLE = 0

ESP_HF_NETWORK_STATE_AVAILABLE

enum esp_hf_service_type_t
+CIEV Service type

Values:

ESP_HF_SERVICE_TYPE_HOME = 0

ESP_HF_SERVICE_TYPE_ROAMING

enum esp_hf_call_status_t
+CIND call status indicator values

Values:

ESP_HF_CALL_STATUS_NO_CALLS = 0
no call in progress

ESP_HF_CALL_STATUS_CALL_IN_PROGRESS = 1
call is present(active or held)

enum esp_hf_call_setup_status_t
+CIND call setup status indicator values

Values:

ESP_HF_CALL_SETUP_STATUS_IDLE = 0
no call setup in progress

ESP_HF_CALL_SETUP_STATUS_INCOMING = 1
incoming call setup in progress

ESP_HF_CALL_SETUP_STATUS_OUTGOING_DIALING = 2
outgoing call setup in dialing state

ESP_HF_CALL_SETUP_STATUS_OUTGOING_ALERTING = 3
outgoing call setup in alerting state

enum esp_hf_roaming_status_t
+CIND roaming status indicator values

Values:

ESP_HF_ROAMING_STATUS_INACTIVE = 0
roaming is not active

ESP_HF_ROAMING_STATUS_ACTIVE
a roaming is active

enum esp_hf_call_held_status_t
+CIND call held indicator values

Values:

ESP_HF_CALL_HELD_STATUS_NONE = 0
no calls held

ESP_HF_CALL_HELD_STATUS_HELD_AND_ACTIVE = 1
both active and held call

ESP_HF_CALL_HELD_STATUS_HELD = 2
call on hold, no active call

enum esp_hf_current_call_status_t
+CLCC status of the call

Values:

ESP_HF_CURRENT_CALL_STATUS_ACTIVE = 0
active

ESP_HF_CURRENT_CALL_STATUS_HELD = 1
held

ESP_HF_CURRENT_CALL_STATUS_DIALING = 2
dialing (outgoing calls only)

ESP_HF_CURRENT_CALL_STATUS_ALERTING = 3
alerting (outgoing calls only)

ESP_HF_CURRENT_CALL_STATUS_INCOMING = 4
incoming (incoming calls only)

ESP_HF_CURRENT_CALL_STATUS_WAITING = 5
waiting (incoming calls only)

ESP_HF_CURRENT_CALL_STATUS_HELD_BY_RESP_HOLD = 6
call held by response and hold

enum esp_hf_current_call_direction_t
+CLCC direction of the call

Values:

ESP_HF_CURRENT_CALL_DIRECTION_OUTGOING = 0
outgoing

ESP_HF_CURRENT_CALL_DIRECTION_INCOMING = 1
incoming

enum esp_hf_current_call_mpty_type_t
+CLCC multi-party call flag

Values:

ESP_HF_CURRENT_CALL_MPTY_TYPE_SINGLE = 0
not a member of a multi-party call

ESP_HF_CURRENT_CALL_MPTY_TYPE_MULTI = 1
member of a multi-party call

enum esp_hf_current_call_mode_t
+CLCC call mode

Values:

ESP_HF_CURRENT_CALL_MODE_VOICE = 0

ESP_HF_CURRENT_CALL_MODE_DATA = 1

ESP_HF_CURRENT_CALL_MODE_FAX = 2

enum esp_hf_call_addr_type_t
+CLCC address type

Values:

ESP_HF_CALL_ADDR_TYPE_UNKNOWN = 0x81
unknown address type

ESP_HF_CALL_ADDR_TYPE_INTERNATIONAL = 0x91
international address

enum esp_hf_subscriber_service_type_t
+CNUM service type of the phone number

Values:

ESP_HF_SUBSCRIBER_SERVICE_TYPE_UNKNOWN = 0
unknown

ESP_HF_SUBSCRIBER_SERVICE_TYPE_VOICE
voice service

ESP_HF_SUBSCRIBER_SERVICE_TYPE_FAX
fax service

enum esp_hf_btrh_status_t
+BTRH response and hold result code

Values:

ESP_HF_BTRH_STATUS_HELD = 0
incoming call is put on hold in AG

ESP_HF_BTRH_STATUS_ACCEPTED
held incoming call is accepted in AG

ESP_HF_BTRH_STATUS_REJECTED
held incoming call is rejected in AG

enum esp_hf_btrh_cmd_t
AT+BTRH response and hold action code.

Values:

ESP_HF_BTRH_CMD_HOLD = 0
put the incoming call on hold

ESP_HF_BTRH_CMD_ACCEPT = 1
accept a held incoming call

ESP_HF_BTRH_CMD_REJECT = 2
reject a held incoming call

enum esp_hf_nrec_t
Values:

ESP_HF_NREC_STOP = 0

ESP_HF_NREC_START

enum esp_hf_call_waiting_status_t

+CCWA response status

Values:

ESP_HF_CALL_WAITING_INACTIVE

ESP_HF_CALL_WAITING_ACTIVE

enum esp_hf_wbs_config_t

Values:

ESP_HF_WBS_NONE

ESP_HF_WBS_NO

ESP_HF_WBS_YES

enum esp_hf_connection_state_t

Bluetooth HFP RFCOMM connection and service level connection status.

Values:

ESP_HF_CONNECTION_STATE_DISCONNECTED = 0

RFCOMM data link channel released

ESP_HF_CONNECTION_STATE_CONNECTING

connecting remote device on the RFCOMM data link

ESP_HF_CONNECTION_STATE_CONNECTED

RFCOMM connection established

ESP_HF_CONNECTION_STATE_SLC_CONNECTED

service level connection established

ESP_HF_CONNECTION_STATE_DISCONNECTING

disconnecting with remote device on the RFCOMM data link

enum esp_hf_chld_type_t

AT+CHLD command values.

Values:

ESP_HF_CHLD_TYPE_REL = 0

<0>, Terminate all held or set UDUB(“busy”) to a waiting call

ESP_HF_CHLD_TYPE_REL_ACC

<1>, Terminate all active calls and accepts a waiting/held call

ESP_HF_CHLD_TYPE_HOLD_ACC

<2>, Hold all active calls and accepts a waiting/held call

ESP_HF_CHLD_TYPE_MERGE

<3>, Add all held calls to a conference

ESP_HF_CHLD_TYPE_MERGE_DETACH

<4>, connect the two calls and disconnects the subscriber from both calls

ESP_HF_CHLD_TYPE_REL_X

<1x>, releases specified calls only

ESP_HF_CHLD_TYPE_PRIV_X

<2x>, request private consultation mode with specified call

enum esp_hf_at_response_code_t

Values:

ESP_HF_AT_RESPONSE_CODE_OK = 0

acknowledges execution of a command line

ESP_HF_AT_RESPONSE_CODE_ERR
command not accepted

ESP_HF_AT_RESPONSE_CODE_NO_CARRIER
connection terminated

ESP_HF_AT_RESPONSE_CODE_BUSY
busy signal detected

ESP_HF_AT_RESPONSE_CODE_NO_ANSWER
connection completion timeout

ESP_HF_AT_RESPONSE_CODE_DELAYED
delayed

ESP_HF_AT_RESPONSE_CODE_BLACKLISTED
blacklisted

ESP_HF_AT_RESPONSE_CODE_CME
CME error

enum esp_hf_at_response_t

Values:

ESP_HF_AT_RESPONSE_ERROR = 0

ESP_HF_AT_RESPONSE_OK

enum esp_hf_cme_err_t

Extended Audio Gateway Error Result Code Response.

Values:

ESP_HF_CME_AG_FAILURE = 0
ag failure

ESP_HF_CME_NO_CONNECTION_TO_PHONE = 1
no connection to phone

ESP_HF_CME_OPERATION_NOT_ALLOWED = 3
operation not allowed

ESP_HF_CME_OPERATION_NOT_SUPPORTED = 4
operation not supported

ESP_HF_CME_PH_SIM_PIN_REQUIRED = 5
PH-SIM PIN Required

ESP_HF_CME_SIM_NOT_INSERTED = 10
SIM not inserted

ESP_HF_CME_SIM_PIN_REQUIRED = 11
SIM PIN required

ESP_HF_CME_SIM_PUK_REQUIRED = 12
SIM PUK required

ESP_HF_CME_SIM_FAILURE = 13
SIM failure

ESP_HF_CME_SIM_BUSY = 14
SIM busy

ESP_HF_CME_INCORRECT_PASSWORD = 16
incorrect password

ESP_HF_CME_SIM_PIN2_REQUIRED = 17
SIM PIN2 required

ESP_HF_CME_SIM_PUK2_REQUIRED = 18
SIM PUK2 required

ESP_HF_CME_MEMEORY_FULL = 20
memory full

ESP_HF_CME_INVALID_INDEX = 21
invalid index

ESP_HF_CME_MEMEORY_FAILURE = 23
memory failure

ESP_HF_CME_TEXT_STRING_TOO_LONG = 24
test string too long

ESP_HF_CME_INVALID_CHARACTERS_IN_TEXT_STRING = 25
invalid characters in text string

ESP_HF_CME_DIAL_STRING_TOO_LONG = 26
dial string too long

ESP_HF_CME_INVALID_CHARACTERS_IN_DIAL_STRING = 27
invalid characters in dial string

ESP_HF_CME_NO_NETWORK_SERVICE = 30
no network service

ESP_HF_CME_NETWORK_TIMEOUT = 31
network timeout

ESP_HF_CME_NETWORK_NOT_ALLOWED = 32
network not allowed emergency calls only

HFP CLIENT API

[Overview](#) [Instructions](#)

API Reference

Header File

- [bt/host/bluedroid/api/include/api/esp_hf_client_api.h](#)

Functions

esp_err_t **esp_hf_client_register_callback** (*esp_hf_client_cb_t* callback)

Register application callback function to HFP client module. This function should be called only after `esp_bluedroid_enable()` completes successfully, used by HFP client.

Return

- `ESP_OK`: success
- `ESP_INVALID_STATE`: if bluetooth stack is not yet enabled
- `ESP_FAIL`: if callback is a NULL function pointer

Parameters

- [in] `callback`: HFP client event callback function

esp_err_t **esp_hf_client_init** (void)

Initialize the bluetooth HFP client module. This function should be called after `esp_bluedroid_enable()` completes successfully.

Return

- `ESP_OK`: if the initialization request is sent successfully
- `ESP_INVALID_STATE`: if bluetooth stack is not yet enabled
- `ESP_FAIL`: others

esp_err_t esp_hf_client_deinit (void)

De-initialize for HFP client module. This function should be called only after `esp_bluetooth_enable()` completes successfully.

Return

- ESP_OK: success
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t esp_hf_client_connect (*esp_bd_addr_t remote_bda*)

Connect to remote bluetooth HFP audio gateway(AG) device, must after `esp_hf_client_init()`

Return

- ESP_OK: connect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- [in] `remote_bda`: remote bluetooth device address

esp_err_t esp_hf_client_disconnect (*esp_bd_addr_t remote_bda*)

Disconnect from the remote HFP audio gateway.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- [in] `remote_bda`: remote bluetooth device address

esp_err_t esp_hf_client_connect_audio (*esp_bd_addr_t remote_bda*)

Create audio connection with remote HFP AG. As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- [in] `remote_bda`: remote bluetooth device address

esp_err_t esp_hf_client_disconnect_audio (*esp_bd_addr_t remote_bda*)

Release the established audio connection with remote HFP AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- [in] `remote_bda`: remote bluetooth device address

esp_err_t esp_hf_client_start_voice_recognition (void)

Enable voice recognition in the AG. As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t esp_hf_client_stop_voice_recognition (void)

Disable voice recognition in the AG. As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer

- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_volume_update** (*esp_hf_volume_control_target_t* type, int volume)

Volume synchronization with AG. As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- [in] type: volume control target, speaker or microphone
- [in] volume: gain of the speaker of microphone, ranges 0 to 15

esp_err_t **esp_hf_client_dial** (const char *number)

Place a call with a specified number, if number is NULL, last called number is called. As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- [in] number: number string of the call. If NULL, the last number is called(aka re-dial)

esp_err_t **esp_hf_client_dial_memory** (int location)

Place a call with number specified by location(speed dial). As a precondition, to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- [in] location: location of the number in the memory

esp_err_t **esp_hf_client_send_chld_cmd** (*esp_hf_chld_type_t* chld, int idx)

Send call hold and multiparty commands, or enhanced call control commands(Use AT+CHLD). As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- [in] chld: AT+CHLD call hold and multiparty handling AT command.
- [in] idx: used in Enhanced Call Control Mechanisms, used if chld is ESP_HF_CHLD_TYPE_REL_X or ESP_HF_CHLD_TYPE_PRIV_X

esp_err_t **esp_hf_client_send_btrh_cmd** (*esp_hf_btrh_cmd_t* btrh)

Send response and hold action command(Send AT+BTRH command) As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- [in] btrh: response and hold action to send

esp_err_t **esp_hf_client_answer_call** (void)

Answer an incoming call(send ATA command). As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

***esp_err_t* esp_hf_client_reject_call** (void)

Reject an incoming call(send AT+CHUP command), As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

***esp_err_t* esp_hf_client_query_current_calls** (void)

Query list of current calls in AG(send AT+CLCC command), As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

***esp_err_t* esp_hf_client_query_current_operator_name** (void)

Query the name of currently selected network operator in AG(use AT+COPS commands) As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

***esp_err_t* esp_hf_client_retrieve_subscriber_info** (void)

Get subscriber information number from AG(send AT+CNUM command) As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

***esp_err_t* esp_hf_client_send_dtmf** (char code)

Transmit DTMF codes during an ongoing call(use AT+VTS commands) As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- [in] code: dtmf code, single ascii character in the set 0-9, #, *, A-D

***esp_err_t* esp_hf_client_request_last_voice_tag_number** (void)

Request a phone number from AG corresponding to last voice tag recorded (send AT+BINP command). As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

***esp_err_t* esp_hf_client_send_nrec** (void)

Disable echo cancellation and noise reduction in the AG (use AT+NREC=0 command) As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: NREC=0 request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

esp_err_t **esp_hf_client_register_data_callback** (*esp_hf_client_incoming_data_cb_t* *recv*,
esp_hf_client_outgoing_data_cb_t *send*)

Register HFP client data output function; the callback is only used in the case that Voice Over HCI is enabled.

Return

- ESP_OK: success
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: if callback is a NULL function pointer

Parameters

- [in] *recv*: HFP client incoming data callback function
- [in] *send*: HFP client outgoing data callback function

void **esp_hf_client_outgoing_data_ready** (void)

Trigger the lower-layer to fetch and send audio data. This function is only used in the case that Voice Over HCI is enabled. Precondition is that the HFP audio connection is connected. After this function is called, lower layer will invoke *esp_hf_client_outgoing_data_cb_t* to fetch data.

void **esp_hf_client_pcm_resample_init** (uint32_t *src_sps*, uint32_t *bits*, uint32_t *channels*)

Initialize the down sampling converter. This is a utility function that can only be used in the case that Voice Over HCI is enabled.

Parameters

- [in] *src_sps*: original samples per second(source audio data, i.e. 48000, 32000, 16000, 44100, 22050, 11025)
- [in] *bits*: number of bits per pcm sample (16)
- [in] *channels*: number of channels (i.e. mono(1), stereo(2)...))

void **esp_hf_client_pcm_resample_deinit** (void)

Deinitialize the down sampling converter.

int32_t **esp_hf_client_pcm_resample** (void **src*, uint32_t *in_bytes*, void **dst*)

Down sampling utility to convert high sampling rate into 8K/16bits 1-channel mode PCM samples. This can only be used in the case that Voice Over HCI is enabled.

Return number of samples converted

Parameters

- [in] *src*: pointer to the buffer where the original sampling PCM are stored
- [in] *in_bytes*: length of the input PCM sample buffer in byte
- [in] *dst*: pointer to the buffer which is to be used to store the converted PCM samples

Unions

union **esp_hf_client_cb_param_t**

#include <esp_hf_client_api.h> HFP client callback parameters.

Public Members

struct *esp_hf_client_cb_param_t::hf_client_conn_stat_param* **conn_stat**
HF callback param of ESP_HF_CLIENT_CONNECTION_STATE_EVT

struct *esp_hf_client_cb_param_t::hf_client_audio_stat_param* **audio_stat**
HF callback param of ESP_HF_CLIENT_AUDIO_STATE_EVT

struct *esp_hf_client_cb_param_t::hf_client_bvra_param* **bvra**
HF callback param of ESP_HF_CLIENT_BVRA_EVT

struct *esp_hf_client_cb_param_t::hf_client_service_availability_param* **service_availability**
HF callback param of ESP_HF_CLIENT_CIND_SERVICE_AVAILABILITY_EVT

```

struct esp_hf_client_cb_param_t::hf_client_network_roaming_param roaming
    HF callback param of ESP_HF_CLIENT_CIND_ROAMING_STATUS_EVT

struct esp_hf_client_cb_param_t::hf_client_signal_strength_ind_param signal_strength
    HF callback param of ESP_HF_CLIENT_CIND_SIGNAL_STRENGTH_EVT

struct esp_hf_client_cb_param_t::hf_client_battery_level_ind_param battery_level
    HF callback param of ESP_HF_CLIENT_CIND_BATTERY_LEVEL_EVT

struct esp_hf_client_cb_param_t::hf_client_current_operator_param cops
    HF callback param of ESP_HF_CLIENT_COPS_CURRENT_OPERATOR_EVT

struct esp_hf_client_cb_param_t::hf_client_call_ind_param call
    HF callback param of ESP_HF_CLIENT_CIND_CALL_EVT

struct esp_hf_client_cb_param_t::hf_client_call_setup_ind_param call_setup
    HF callback param of ESP_HF_CLIENT_BVRA_EVT

struct esp_hf_client_cb_param_t::hf_client_call_held_ind_param call_held
    HF callback param of ESP_HF_CLIENT_CIND_CALL_HELD_EVT

struct esp_hf_client_cb_param_t::hf_client_btrh_param btrh
    HF callback param of ESP_HF_CLIENT_BRTH_EVT

struct esp_hf_client_cb_param_t::hf_client_clip_param clip
    HF callback param of ESP_HF_CLIENT_CLIP_EVT

struct esp_hf_client_cb_param_t::hf_client_ccwa_param ccwa
    HF callback param of ESP_HF_CLIENT_BVRA_EVT

struct esp_hf_client_cb_param_t::hf_client_clcc_param clcc
    HF callback param of ESP_HF_CLIENT_CLCC_EVT

struct esp_hf_client_cb_param_t::hf_client_volume_control_param volume_control
    HF callback param of ESP_HF_CLIENT_VOLUME_CONTROL_EVT

struct esp_hf_client_cb_param_t::hf_client_at_response_param at_response
    HF callback param of ESP_HF_CLIENT_AT_RESPONSE_EVT

struct esp_hf_client_cb_param_t::hf_client_cnum_param cnum
    HF callback param of ESP_HF_CLIENT_CNUM_EVT

struct esp_hf_client_cb_param_t::hf_client_bsirparam bsir
    HF callback param of ESP_HF_CLIENT_BSIR_EVT

struct esp_hf_client_cb_param_t::hf_client_binp_param binp
    HF callback param of ESP_HF_CLIENT_BINP_EVT

struct hf_client_at_response_param
    #include <esp_hf_client_api.h> ESP_HF_CLIENT_AT_RESPONSE_EVT.

```

Public Members

esp_hf_at_response_code_t **code**

AT response code

esp_hf_cme_err_t **cme**

Extended Audio Gateway Error Result Code

struct **hf_client_audio_stat_param**

#include <esp_hf_client_api.h> ESP_HF_CLIENT_AUDIO_STATE_EVT.

Public Members

esp_hf_client_audio_state_t **state**
audio connection state

esp_bd_addr_t **remote_bda**
remote bluetooth device address

struct hf_client_battery_level_ind_param
#include <esp_hf_client_api.h> ESP_HF_CLIENT_CIND_BATTERY_LEVEL_EVT.

Public Members

int value
battery charge value, ranges from 0 to 5

struct hf_client_binp_param
#include <esp_hf_client_api.h> ESP_HF_CLIENT_BINP_EVT.

Public Members

const char *number
phone number corresponding to the last voice tag in the HF

struct hf_client_bsirparam
#include <esp_hf_client_api.h> ESP_HF_CLIENT_BSIR_EVT.

Public Members

esp_hf_client_in_band_ring_state_t **state**
setting state of in-band ring tone

struct hf_client_btrh_param
#include <esp_hf_client_api.h> ESP_HF_CLIENT_BTRH_EVT.

Public Members

esp_hf_btrh_status_t **status**
call hold and response status result code

struct hf_client_bvra_param
#include <esp_hf_client_api.h> ESP_HF_CLIENT_BVRA_EVT.

Public Members

esp_hf_vr_state_t **value**
voice recognition state

struct hf_client_call_held_ind_param
#include <esp_hf_client_api.h> ESP_HF_CLIENT_CIND_CALL_HELD_EVT.

Public Members

esp_hf_call_held_status_t **status**
bluetooth proprietary call hold status indicator

struct hf_client_call_ind_param
#include <esp_hf_client_api.h> ESP_HF_CLIENT_CIND_CALL_EVT.

Public Members

esp_hf_call_status_t **status**
call status indicator

struct hf_client_call_setup_ind_param
#include <esp_hf_client_api.h> ESP_HF_CLIENT_CIND_CALL_SETUP_EVT.

Public Members

esp_hf_call_setup_status_t **status**
call setup status indicator

struct hf_client_ccwa_param
#include <esp_hf_client_api.h> ESP_HF_CLIENT_CCWA_EVT.

Public Members

const char *number
phone number string of waiting call

struct hf_client_clcc_param
#include <esp_hf_client_api.h> ESP_HF_CLIENT_CLCC_EVT.

Public Members

int idx
numbering(starting with 1) of the call

esp_hf_current_call_direction_t **dir**
direction of the call

esp_hf_current_call_status_t **status**
status of the call

esp_hf_current_call_mpty_type_t **mpty**
multi-party flag

char *number
phone number(optional)

struct hf_client_clip_param
#include <esp_hf_client_api.h> ESP_HF_CLIENT_CLIP_EVT.

Public Members

const char *number
phone number string of call

```
struct hf_client_cnum_param  
#include <esp_hf_client_api.h> ESP_HF_CLIENT_CNUM_EVT.
```

Public Members

```
const char *number  
    phone number string  
  
esp_hf_subscriber_service_type_t type  
    service type that the phone number relates to
```

```
struct hf_client_conn_stat_param  
#include <esp_hf_client_api.h> ESP_HF_CLIENT_CONNECTION_STATE_EVT.
```

Public Members

```
esp_hf_client_connection_state_t state  
    HF connection state  
  
uint32_t peer_feat  
    AG supported features  
  
uint32_t chld_feat  
    AG supported features on call hold and multiparty services  
  
esp_bd_addr_t remote_bda  
    remote bluetooth device address
```

```
struct hf_client_current_operator_param  
#include <esp_hf_client_api.h> ESP_HF_CLIENT_COPS_CURRENT_OPERATOR_EVT.
```

Public Members

```
const char *name  
    name of the network operator
```

```
struct hf_client_network_roaming_param  
#include <esp_hf_client_api.h> ESP_HF_CLIENT_CIND_ROAMING_STATUS_EVT.
```

Public Members

```
esp_hf_roaming_status_t status  
    roaming status
```

```
struct hf_client_service_availability_param  
#include <esp_hf_client_api.h> ESP_HF_CLIENT_CIND_SERVICE_AVAILABILITY_EVT.
```

Public Members

```
esp_hf_network_state_t status  
    service availability status
```

```
struct hf_client_signal_strength_ind_param  
#include <esp_hf_client_api.h> ESP_HF_CLIENT_CIND_SIGNAL_STRENGTH_EVT.
```

Public Members

int **value**
signal strength value, ranges from 0 to 5

struct hf_client_volume_control_param
#include <esp_hf_client_api.h> ESP_HF_CLIENT_VOLUME_CONTROL_EVT.

Public Members

esp_hf_volume_control_target_t **type**
volume control target, speaker or microphone

int **volume**
gain, ranges from 0 to 15

Macros

ESP_BT_HF_CLIENT_NUMBER_LEN
ESP_BT_HF_CLIENT_OPERATOR_NAME_LEN
ESP_HF_CLIENT_PEER_FEAT_3WAY
ESP_HF_CLIENT_PEER_FEAT_ECNR
ESP_HF_CLIENT_PEER_FEAT_VREC
ESP_HF_CLIENT_PEER_FEAT_INBAND
ESP_HF_CLIENT_PEER_FEAT_VTAG
ESP_HF_CLIENT_PEER_FEAT_REJECT
ESP_HF_CLIENT_PEER_FEAT_ECS
ESP_HF_CLIENT_PEER_FEAT_ECC
ESP_HF_CLIENT_PEER_FEAT_EXTEERR
ESP_HF_CLIENT_PEER_FEAT_CODEC
ESP_HF_CLIENT_CHLD_FEAT_REL
ESP_HF_CLIENT_CHLD_FEAT_REL_ACC
ESP_HF_CLIENT_CHLD_FEAT_REL_X
ESP_HF_CLIENT_CHLD_FEAT_HOLD_ACC
ESP_HF_CLIENT_CHLD_FEAT_PRIV_X
ESP_HF_CLIENT_CHLD_FEAT_MERGE
ESP_HF_CLIENT_CHLD_FEAT_MERGE_DETACH

Type Definitions

typedef void (*esp_hf_client_incoming_data_cb_t) (const uint8_t *buf, uint32_t len)
HFP client incoming data callback function, the callback is useful in case of Voice Over HCI.

Parameters

- [in] **buf**: : pointer to incoming data(payload of HCI synchronous data packet), the buffer is allocated inside bluetooth protocol stack and will be released after invoke of the callback is finished.
- [in] **len**: : size(in bytes) in buf

```
typedef uint32_t (*esp_hf_client_outgoing_data_cb_t) (uint8_t *buf, uint32_t len)
```

HFP client outgoing data callback function, the callback is useful in case of Voice Over HCI. Once audio connection is set up and the application layer has prepared data to send, the lower layer will call this function to read data and then send. This callback is supposed to be implemented as non-blocking, and if data is not enough, return value 0 is supposed.

Return length of data successfully read

Parameters

- [in] `buf`: : pointer to incoming data(payload of HCI synchronous data packet), the buffer is allocated inside bluetooth protocol stack and will be released after invoke of the callback is finished.
- [in] `len`: : size(in bytes) in buf

```
typedef void (*esp_hf_client_cb_t) (esp_hf_client_cb_event_t event, esp_hf_client_cb_param_t *param)
```

HFP client callback function type.

Parameters

- `event`: : Event type
- `param`: : Pointer to callback parameter

Enumerations

```
enum esp_hf_client_connection_state_t
```

Bluetooth HFP RFCOMM connection and service level connection status.

Values:

```
ESP_HF_CLIENT_CONNECTION_STATE_DISCONNECTED = 0  
RFCOMM data link channel released
```

```
ESP_HF_CLIENT_CONNECTION_STATE_CONNECTING  
connecting remote device on the RFCOMM data link
```

```
ESP_HF_CLIENT_CONNECTION_STATE_CONNECTED  
RFCOMM connection established
```

```
ESP_HF_CLIENT_CONNECTION_STATE_SLC_CONNECTED  
service level connection established
```

```
ESP_HF_CLIENT_CONNECTION_STATE_DISCONNECTING  
disconnecting with remote device on the RFCOMM dat link
```

```
enum esp_hf_client_audio_state_t
```

Bluetooth HFP audio connection status.

Values:

```
ESP_HF_CLIENT_AUDIO_STATE_DISCONNECTED = 0  
audio connection released
```

```
ESP_HF_CLIENT_AUDIO_STATE_CONNECTING  
audio connection has been initiated
```

```
ESP_HF_CLIENT_AUDIO_STATE_CONNECTED  
audio connection is established
```

```
ESP_HF_CLIENT_AUDIO_STATE_CONNECTED_MSBC  
mSBC audio connection is established
```

```
enum esp_hf_client_in_band_ring_state_t
```

in-band ring tone state

Values:

```
ESP_HF_CLIENT_IN_BAND_RINGTONE_NOT_PROVIDED = 0
```

```
ESP_HF_CLIENT_IN_BAND_RINGTONE_PROVIDED
```

enum esp_hf_client_cb_event_t

HF CLIENT callback events.

Values:

- ESP_HF_CLIENT_CONNECTION_STATE_EVT = 0**
connection state changed event
- ESP_HF_CLIENT_AUDIO_STATE_EVT**
audio connection state change event
- ESP_HF_CLIENT_BVRA_EVT**
voice recognition state change event
- ESP_HF_CLIENT_CIND_CALL_EVT**
call indication
- ESP_HF_CLIENT_CIND_CALL_SETUP_EVT**
call setup indication
- ESP_HF_CLIENT_CIND_CALL_HELD_EVT**
call held indication
- ESP_HF_CLIENT_CIND_SERVICE_AVAILABILITY_EVT**
network service availability indication
- ESP_HF_CLIENT_CIND_SIGNAL_STRENGTH_EVT**
signal strength indication
- ESP_HF_CLIENT_CIND_ROAMING_STATUS_EVT**
roaming status indication
- ESP_HF_CLIENT_CIND_BATTERY_LEVEL_EVT**
battery level indication
- ESP_HF_CLIENT_COPS_CURRENT_OPERATOR_EVT**
current operator information
- ESP_HF_CLIENT_BTRH_EVT**
call response and hold event
- ESP_HF_CLIENT_CLIP_EVT**
Calling Line Identification notification
- ESP_HF_CLIENT_CCWA_EVT**
call waiting notification
- ESP_HF_CLIENT_CLCC_EVT**
list of current calls notification
- ESP_HF_CLIENT_VOLUME_CONTROL_EVT**
audio volume control command from AG, provided by +VGM or +VGS message
- ESP_HF_CLIENT_AT_RESPONSE_EVT**
AT command response event
- ESP_HF_CLIENT_CNUM_EVT**
subscriber information response from AG
- ESP_HF_CLIENT_BSIR_EVT**
setting of in-band ring tone
- ESP_HF_CLIENT_BINP_EVT**
requested number of last voice tag from AG
- ESP_HF_CLIENT_RING_IND_EVT**
ring indication event

HFP AG API

Overview [Instructions](#)

API Reference

Header File

- `bt/host/bluedroid/api/include/api/esp_hf_ag_api.h`

Functions

esp_err_t **esp_bt_hf_register_callback** (*esp_hf_cb_t* callback)

Register application callback function to HFP AG module. This function should be called only after `esp_bluedroid_enable()` completes successfully, used by HFP AG.

Return

- `ESP_OK`: success
- `ESP_INVALID_STATE`: if bluetooth stack is not yet enabled
- `ESP_FAIL`: if callback is a NULL function pointer

Parameters

- [in] `callback`: HFP AG event callback function

esp_err_t **esp_bt_hf_init** (*esp_bd_addr_t* remote_addr)

Initialize the bluetooth HF AG module. This function should be called after `esp_bluedroid_enable()` completes successfully.

Return

- `ESP_OK`: if the initialization request is sent successfully
- `ESP_INVALID_STATE`: if bluetooth stack is not yet enabled
- `ESP_FAIL`: others

Parameters

- [in] `remote_addr`: remote bluetooth device address

esp_err_t **esp_bt_hf_deinit** (*esp_bd_addr_t* remote_addr)

De-initialize for HF AG module. This function should be called only after `esp_bluedroid_enable()` completes successfully.

Return

- `ESP_OK`: success
- `ESP_INVALID_STATE`: if bluetooth stack is not yet enabled
- `ESP_FAIL`: others

Parameters

- [in] `remote_addr`: remote bluetooth device address

esp_err_t **esp_bt_hf_connect** (*esp_bd_addr_t* remote_bda)

Connect to remote bluetooth HFP client device, must after `esp_bt_hf_init()`

Return

- `ESP_OK`: connect request is sent to lower layer
- `ESP_INVALID_STATE`: if bluetooth stack is not yet enabled
- `ESP_FAIL`: others

Parameters

- [in] `remote_bda`: remote bluetooth HFP client device address

esp_err_t **esp_bt_hf_disconnect** (*esp_bd_addr_t* remote_bda)

Disconnect from the remote HFP client.

Return

- `ESP_OK`: disconnect request is sent to lower layer
- `ESP_INVALID_STATE`: if bluetooth stack is not yet enabled
- `ESP_FAIL`: others

Parameters

- [in] remote_bda: remote bluetooth device address

esp_err_t esp_bt_hf_connect_audio(esp_bd_addr_t remote_bda)

Create audio connection with remote HFP client. As a precondition to use this API, Service Level Connection shall exist between HF client and AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- [in] remote_bda: remote bluetooth device address

esp_err_t esp_bt_hf_disconnect_audio(esp_bd_addr_t remote_bda)

Release the established audio connection with remote HFP client.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- [in] remote_bda: remote bluetooth device address

esp_err_t esp_bt_hf_vra(esp_bd_addr_t remote_bda, esp_hf_vr_state_t value)

Response of Volume Recognition Command(AT+VRA) from HFP client. As a precondition to use this API, Service Level Connection shall exist with HFP client.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- [in] remote_bda: the device address of voice recognition initiator
- [in] value: 0 - voice recognition disabled, 1- voice recognition enabled

esp_err_t esp_bt_hf_volume_control(esp_bd_addr_t remote_bda, esp_hf_volume_control_target_t type, int volume)

Volume synchronization with HFP client. As a precondition to use this API, Service Level Connection shall exist with HFP client.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- [in] remote_bda: remote bluetooth device address
- [in] type: volume control target, speaker or microphone
- [in] volume: gain of the speaker of microphone, ranges 0 to 15

*esp_err_t esp_hf_unat_response(esp_bd_addr_t remote_addr, char *unat)*

Handle Unknown AT command from HFP Client. As a precondition to use this API, Service Level Connection shall exist between AG and HF Client.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- [in] remote_addr: remote bluetooth device address
- [in] unat: User AT command response to HF Client. It will response "ERROR" by default if unat is NULL.

esp_err_t **esp_bt_hf_cmee_response** (*esp_bd_addr_t* remote_bda, *esp_hf_at_response_code_t* response_code, *esp_hf_cme_err_t* error_code)

Unsolicited send extend AT error code to HFP Client. As a precondition to use this API, Service Level Connection shall exist between AG and HF Client.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- [in] remote_bda: remote bluetooth device address
- [in] response_code: AT command response code
- [in] error_code: CME error code

esp_err_t **esp_bt_hf_indchange_notification** (*esp_bd_addr_t* remote_addr, *esp_hf_call_status_t* call_state, *esp_hf_call_setup_status_t* call_setup_state, *esp_hf_network_state_t* ntk_state, int signal)

Unsolicited send device status notification to HFP Client. As a precondition to use this API, Service Level Connection shall exist between AG and HF Client.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- [in] remote_addr: remote bluetooth device address
- [in] call_state: call state
- [in] call_setup_state: call setup state
- [in] ntk_state: network service state
- [in] signal: signal strength from 0 to 5

esp_err_t **esp_bt_hf_cind_response** (*esp_bd_addr_t* remote_addr, *esp_hf_call_status_t* call_state, *esp_hf_call_setup_status_t* call_setup_state, *esp_hf_network_state_t* ntk_state, int signal, *esp_hf_roaming_status_t* roam, int batt_lev, *esp_hf_call_held_status_t* call_held_status)

Response to device individual indicators to HFP Client. As a precondition to use this API, Service Level Connection shall exist between AG and HF Client.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- [in] remote_addr: remote bluetooth device address
- [in] call_state: call state
- [in] call_setup_state: call setup state
- [in] ntk_state: network service state
- [in] signal: signal strength from 0 to 5
- [in] roam: roam state
- [in] batt_lev: battery level from 0 to 5
- [in] call_held_status: call held status

esp_err_t **esp_bt_hf_cops_response** (*esp_bd_addr_t* remote_addr, char *name)

Response for AT+COPS command from HF Client. As a precondition to use this API, Service Level Connection shall exist with HFP Client.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- [in] remote_addr: remote bluetooth device address
- [in] name: current operator name

esp_err_t **esp_bt_hf_clcc_response** (*esp_bd_addr_t* remote_addr, int index, *esp_hf_current_call_direction_t* dir, *esp_hf_current_call_status_t* current_call_state, *esp_hf_current_call_mode_t* mode, *esp_hf_current_call_mpty_type_t* mpty, char *number, *esp_hf_call_addr_type_t* type)

Response to AT+CLCC command from HFP Client. As a precondition to use this API, Service Level Connection shall exist between AG and HF Client.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- [in] remote_addr: remote bluetooth device address
- [in] index: the index of current call
- [in] dir: call direction (incoming/outgoing)
- [in] current_call_state: current call state
- [in] mode: current call mode (voice/data/fax)
- [in] mpty: single or multi type
- [in] number: current call number
- [in] type: international type or unknow

esp_err_t **esp_bt_hf_cnum_response** (*esp_bd_addr_t* remote_addr, char *number, *esp_hf_subscriber_service_type_t* type)

Response for AT+CNUM command from HF Client. As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- [in] remote_addr: remote bluetooth device address
- [in] number: registration number
- [in] type: service type (unknown/voice/fax)

esp_err_t **esp_bt_hf_bsir** (*esp_bd_addr_t* remote_addr, *esp_hf_in_band_ring_state_t* state)

Inform HF Client that AG Provided in-band ring tone or not. As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- [in] remote_addr: remote bluetooth device address
- [in] state: in-band ring tone state

esp_err_t **esp_bt_hf_answer_call** (*esp_bd_addr_t* remote_addr, int num_active, int num_held, *esp_hf_call_status_t* call_state, *esp_hf_call_setup_status_t* call_setup_state, char *number, *esp_hf_call_addr_type_t* call_addr_type)

Answer Incoming Call from AG. As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled

- ESP_FAIL: others

Parameters

- [in] remote_addr: remote bluetooth device address
- [in] num_active: the number of active call
- [in] num_held: the number of held call
- [in] call_state: call state
- [in] call_setup_state: call setup state
- [in] number: number of the incoming call
- [in] call_addr_type: call address type

esp_err_t **esp_bt_hf_reject_call** (*esp_bd_addr_t* remote_addr, int num_active, int num_held, *esp_hf_call_status_t* call_state, *esp_hf_call_setup_status_t* call_setup_state, char *number, *esp_hf_call_addr_type_t* call_addr_type)

Reject Incoming Call from AG. As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- [in] remote_addr: remote bluetooth device address
- [in] num_active: the number of active call
- [in] num_held: the number of held call
- [in] call_state: call state
- [in] call_setup_state: call setup state
- [in] number: number of the incoming call
- [in] call_addr_type: call address type

esp_err_t **esp_bt_hf_out_call** (*esp_bd_addr_t* remote_addr, int num_active, int num_held, *esp_hf_call_status_t* call_state, *esp_hf_call_setup_status_t* call_setup_state, char *number, *esp_hf_call_addr_type_t* call_addr_type)

Reject incoming call from AG. As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- [in] remote_addr: remote bluetooth device address
- [in] num_active: the number of active call
- [in] num_held: the number of held call
- [in] call_state: call state
- [in] call_setup_state: call setup state
- [in] number: number of the outgoing call
- [in] call_addr_type: call address type

esp_err_t **esp_bt_hf_end_call** (*esp_bd_addr_t* remote_addr, int num_active, int num_held, *esp_hf_call_status_t* call_state, *esp_hf_call_setup_status_t* call_setup_state, char *number, *esp_hf_call_addr_type_t* call_addr_type)

End an ongoing call. As a precondition to use this API, Service Level Connection shall exist with AG.

Return

- ESP_OK: disconnect request is sent to lower layer
- ESP_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- [in] remote_addr: remote bluetooth device address

- [in] `num_active`: the number of active call
- [in] `num_held`: the number of held call
- [in] `call_state`: call state
- [in] `call_setup_state`: call setup state
- [in] `number`: number of the call
- [in] `call_addr_type`: call address type

`esp_err_t esp_bt_hf_register_data_callback` (`esp_hf_incoming_data_cb_t` *recv*,
`esp_hf_outgoing_data_cb_t` *send*)

Register AG data output function; the callback is only used in the case that Voice Over HCI is enabled.

Return

- `ESP_OK`: success
- `ESP_INVALID_STATE`: if bluetooth stack is not yet enabled
- `ESP_FAIL`: if callback is a NULL function pointer

Parameters

- [in] `recv`: HFP client incoming data callback function
- [in] `send`: HFP client outgoing data callback function

void `esp_hf_outgoing_data_ready` (void)

Trigger the lower-layer to fetch and send audio data. This function is only used in the case that Voice Over HCI is enabled. Precondition is that the HFP audio connection is connected. After this function is called, lower layer will invoke `esp_hf_client_outgoing_data_cb_t` to fetch data.

Unions

`union esp_hf_cb_param_t`

`#include <esp_hf_ag_api.h>` HFP AG callback parameters.

Public Members

```

struct esp_hf_cb_param_t::hf_conn_stat_param conn_stat
    AG callback param of ESP_HF_CONNECTION_STATE_EVT

struct esp_hf_cb_param_t::hf_audio_stat_param audio_stat
    AG callback param of ESP_HF_AUDIO_STATE_EVT

struct esp_hf_cb_param_t::hf_vra_rep_param vra_rep
    AG callback param of ESP_HF_BVRA_RESPONSE_EVT

struct esp_hf_cb_param_t::hf_volume_control_param volume_control
    AG callback param of ESP_HF_VOLUME_CONTROL_EVT

struct esp_hf_cb_param_t::hf_unat_rep_param unat_rep
    AG callback param of ESP_HF_UNAT_RESPONSE_EVT

struct esp_hf_cb_param_t::hf_cind_param cind
    AG callback param of ESP_HF_CIND_RESPONSE_EVT

struct esp_hf_cb_param_t::hf_out_call_param out_call
    AG callback param of ESP_HF_DIAL_EVT

struct esp_hf_cb_param_t::hf_vts_rep_param vts_rep
    AG callback param of ESP_HF_VTS_RESPONSE_EVT

struct esp_hf_cb_param_t::hf_nrec_param nrec
    AG callback param of ESP_HF_NREC_RESPONSE_EVT

struct esp_hf_cb_param_t::hf_wbs_rep_param wbs_rep
    AG callback param of ESP_HF_WBS_RESPONSE_EVT

struct esp_hf_cb_param_t::hf_bcs_rep_param bcs_rep
    AG callback param of ESP_HF_BCS_RESPONSE_EVT

```

```
struct hf_audio_stat_param  
#include <esp_hf_ag_api.h> ESP_HF_AUDIO_STATE_EVT.
```

Public Members

esp_bd_addr_t **remote_addr**
remote bluetooth device address

esp_hf_audio_state_t **state**
audio connection state

```
struct hf_bcs_rep_param  
#include <esp_hf_ag_api.h> ESP_HF_BCS_RESPONSE_EVT.
```

Public Members

esp_hf_wbs_config_t **mode**
codec mode CVSD or mSBC

```
struct hf_cind_param  
#include <esp_hf_ag_api.h> ESP_HF_CIND_RESPONSE_EVT.
```

Public Members

esp_hf_call_status_t **call_status**
call status indicator

esp_hf_call_setup_status_t **call_setup_status**
call setup status indicator

esp_hf_network_state_t **svc**
bluetooth proprietary call hold status indicator

int **signal_strength**
bluetooth proprietary call hold status indicator

esp_hf_roaming_status_t **roam**
bluetooth proprietary call hold status indicator

int **battery_level**
battery charge value, ranges from 0 to 5

esp_hf_call_held_status_t **call_held_status**
bluetooth proprietary call hold status indicator

```
struct hf_conn_stat_param  
#include <esp_hf_ag_api.h> ESP_HS_CONNECTION_STATE_EVT.
```

Public Members

esp_bd_addr_t **remote_bda**
remote bluetooth device address

esp_hf_connection_state_t **state**
Connection state

uint32_t **peer_feat**
HF supported features

uint32_t **chld_feat**
AG supported features on call hold and multiparty services

```
struct hf_nrec_param  
    #include <esp_hf_ag_api.h> ESP_HF_NREC_RESPOSNE_EVT.
```

Public Members

```
esp_hf_nrec_t state  
    NREC enabled or disabled
```

```
struct hf_out_call_param  
    #include <esp_hf_ag_api.h> ESP_HF_DIAL_EVT.
```

Public Members

```
esp_bd_addr_t remote_addr  
    remote bluetooth device address
```

```
char *num_or_loc  
    location in phone memory
```

```
struct hf_unat_rep_param  
    #include <esp_hf_ag_api.h> ESP_HF_UNAT_RESPOSNE_EVT.
```

Public Members

```
char *unat  
    unknown AT command string
```

```
struct hf_volume_control_param  
    #include <esp_hf_ag_api.h> ESP_HF_VOLUME_CONTROL_EVT.
```

Public Members

```
esp_hf_volume_type_t type  
    volume control target, speaker or microphone
```

```
int volume  
    gain, ranges from 0 to 15
```

```
struct hf_vra_rep_param  
    #include <esp_hf_ag_api.h> ESP_HF_BVRA_RESPONSE_EVT.
```

Public Members

```
esp_bd_addr_t remote_addr  
    remote bluetooth device address
```

```
esp_hf_vr_state_t value  
    voice recognition state
```

```
struct hf_vts_rep_param  
    #include <esp_hf_ag_api.h> ESP_HF_VTS_RESPOSNE_EVT.
```

Public Members

char ***code**
MTF code from HF Client

struct hf_wbs_rep_param
#include <esp_hf_ag_api.h> ESP_HF_WBS_RESPONSE_EVT.

Public Members

esp_hf_wbs_config_t **codec**
codec mode CVSD or mSBC

Macros

ESP_HF_PEER_FEAT_3WAY
ESP_HF_PEER_FEAT_ECNR
ESP_HF_PEER_FEAT_VREC
ESP_HF_PEER_FEAT_INBAND
ESP_HF_PEER_FEAT_VTAG
ESP_HF_PEER_FEAT_REJECT
ESP_HF_PEER_FEAT_ECS
ESP_HF_PEER_FEAT_ECC
ESP_HF_PEER_FEAT_EXTERR
ESP_HF_PEER_FEAT_CODEC
ESP_HF_CHLD_FEAT_REL
ESP_HF_CHLD_FEAT_REL_ACC
ESP_HF_CHLD_FEAT_REL_X
ESP_HF_CHLD_FEAT_HOLD_ACC
ESP_HF_CHLD_FEAT_PRIV_X
ESP_HF_CHLD_FEAT_MERGE
ESP_HF_CHLD_FEAT_MERGE_DETACH

Type Definitions

typedef void (*esp_hf_incoming_data_cb_t) (const uint8_t *buf, uint32_t len)
AG incoming data callback function, the callback is useful in case of Voice Over HCI.

Parameters

- [in] **buf**: : pointer to incoming data(payload of HCI synchronous data packet), the buffer is allocated inside bluetooth protocol stack and will be released after invoke of the callback is finished.
- [in] **len**: : size(in bytes) in buf

typedef uint32_t (*esp_hf_outgoing_data_cb_t) (uint8_t *buf, uint32_t len)

AG outgoing data callback function, the callback is useful in case of Voice Over HCI. Once audio connection is set up and the application layer has prepared data to send, the lower layer will call this function to read data and then send. This callback is supposed to be implemented as non-blocking, and if data is not enough, return value 0 is supposed.

Return length of data successfully read

Parameters

- [in] **buf**: : pointer to incoming data(payload of HCI synchronous data packet), the buffer is allocated inside bluetooth protocol stack and will be released after invoke of the callback is finished.

- [in] len: : size(in bytes) in buf

typedef void (***esp_hf_cb_t**) (*esp_hf_cb_event_t* event, *esp_hf_cb_param_t* *param)
HF AG callback function type.

Parameters

- event: : Event type
- param: : Pointer to callback parameter

Enumerations

enum esp_hf_cb_event_t

HF callback events.

Values:

ESP_HF_CONNECTION_STATE_EVT = 0

Connection state changed event

ESP_HF_AUDIO_STATE_EVT

Audio connection state change event

ESP_HF_BVRA_RESPONSE_EVT

Voice recognition state change event

ESP_HF_VOLUME_CONTROL_EVT

Audio volume control command from HF Client, provided by +VGM or +VGS message

ESP_HF_UNAT_RESPONSE_EVT

Unknown AT cmd Response

ESP_HF_IND_UPDATE_EVT

Indicator Update Event

ESP_HF_CIND_RESPONSE_EVT

Call And Device Indicator Response

ESP_HF_COPS_RESPONSE_EVT

Current operator information

ESP_HF_CLCC_RESPONSE_EVT

List of current calls notification

ESP_HF_CNUM_RESPONSE_EVT

Subscriber information response from HF Client

ESP_HF_VTS_RESPONSE_EVT

Enable or not DTMF

ESP_HF_NREC_RESPONSE_EVT

Enable or not NREC

ESP_HF_ATA_RESPONSE_EVT

Answer an Incoming Call

ESP_HF_CHUP_RESPONSE_EVT

Reject an Incoming Call

ESP_HF_DIAL_EVT

Origin an outgoing call with specific number or the dial the last number

ESP_HF_WBS_RESPONSE_EVT

Codec Status

ESP_HF_BCS_RESPONSE_EVT

Final Codec Choice

2.1.5 NimBLE-based host APIs

Overview

Apache MyNewt NimBLE is a highly configurable and BT SIG qualifiable BLE stack providing both host and controller functionalities. ESP-IDF supports NimBLE host stack which is specifically ported for ESP32 platform and FreeRTOS. The underlying controller is still the same (as in case of Bluedroid) providing VHCI interface. Refer to [NimBLE user guide](#) for a complete list of features and additional information on NimBLE stack. Most features of NimBLE including BLE Mesh are supported by ESP-IDF. The porting layer is kept cleaner by maintaining all the existing APIs of NimBLE along with a single ESP-NimBLE API for initialization, making it simpler for the application developers.

Architecture

Currently, NimBLE host and controller support different transports such as UART and RAM between them. However, RAM transport cannot be used as is in case of ESP as ESP controller supports VHCI interface and buffering schemes used by NimBLE host is incompatible with that used by ESP controller. Therefore, a new transport between NimBLE host and ESP controller has been added. This is depicted in the figure below. This layer is responsible for maintaining pool of transport buffers and formatting buffers exchanges between host and controller as per the requirements.



Fig. 1: ESP NimBLE Stack

Threading Model

The NimBLE host can run inside the application thread or can have its own independent thread. This flexibility is inherently provided by NimBLE design. By default, a thread is spawned by the porting function `nimble_port_freertos_init`. This behavior can be changed by overriding the same function. For BLE Mesh, additional thread (advertising thread) is used which keeps on feeding advertisement events to the main thread.

Programming Sequence

To begin with, make sure that the NimBLE stack is enabled from menuconfig *choose NimBLE for the Bluetooth host*.

Typical programming sequence with NimBLE stack consists of the following steps:

- Initialize NVS flash using `nvs_flash_init()` API. This is because ESP controller uses NVS during initialization.
- Call `esp_nimble_hci_and_controller_init()` to initialize ESP controller as well as transport layer. This will also link the host and controller modules together. Alternatively, if ESP controller is already initialized, then `esp_nimble_hci_init()` can be called for the remaining initialization.
- Initialize the host stack using `nimble_port_init`.
- Initialize the required NimBLE host configuration parameters and callbacks
- Perform application specific tasks/initialization
- Run the thread for host stack using `nimble_port_freertos_init`

This documentation does not cover NimBLE APIs. Refer to [NimBLE tutorial](#) for more details on the programming sequence/NimBLE APIs for different scenarios.

API Reference

Header File

- [bt/host/nimble/esp-hci/include/esp_nimble_hci.h](#)

Functions

esp_err_t **esp_nimble_hci_init** (void)

Initialize VHCI transport layer between NimBLE Host and ESP Bluetooth controller.

This function initializes the transport buffers to be exchanged between NimBLE host and ESP controller. It also registers required host callbacks with the controller.

Return

- ESP_OK if the initialization is successful
- Appropriate error code from *esp_err_t* in case of an error

esp_err_t **esp_nimble_hci_and_controller_init** (void)

Initialize ESP Bluetooth controller(link layer) and VHCI transport layer between NimBLE Host and ESP Bluetooth controller.

This function initializes ESP controller in BLE only mode and the transport buffers to be exchanged between NimBLE host and ESP controller. It also registers required host callbacks with the controller.

Below is the sequence of APIs to be called to init/enable NimBLE host and ESP controller:

```
void ble_host_task(void *param)
{
    nimble_port_run(); //This function will return only when nimble_port_
    ↪stop() is executed.
    nimble_port_freertos_deinit();
}

int ret = esp_nimble_hci_and_controller_init();
if (ret != ESP_OK) {
    ESP_LOGE(TAG, "esp_nimble_hci_and_controller_init() failed with error: %d
    ↪", ret);
    return;
}

nimble_port_init();

//Initialize the NimBLE Host configuration

nimble_port_freertos_init(ble_host_task);
```

nimble_port_freertos_init() is an optional call that creates a new task in which the NimBLE host will run. The task function should have a call to *nimble_port_run*(). If a separate task is not required, calling *nimble_port_run*() will run the NimBLE host in the current task.

Return

- ESP_OK if the initialization is successful
- Appropriate error code from *esp_err_t* in case of an error

esp_err_t **esp_nimble_hci_deinit** (void)

Deinitialize VHCI transport layer between NimBLE Host and ESP Bluetooth controller.

Note This function should be called after the NimBLE host is deinitialized.

Return

- ESP_OK if the deinitialization is successful
- Appropriate error codes from *esp_err_t* in case of an error

esp_err_t **esp_nimble_hci_and_controller_deinit** (void)

Deinitialize VHCI transport layer between NimBLE Host and ESP Bluetooth controller and disable and deinitialize the controller.

Below is the sequence of APIs to be called to disable/deinit NimBLE host and ESP controller:

Note This function should not be executed in the context of Bluetooth host task.

Note This function should be called after the NimBLE host is deinitialized.

```
int ret = nimble_port_stop();
if (ret == 0) {
    nimble_port_deinit();

    ret = esp_nimble_hci_and_controller_deinit();
    if (ret != ESP_OK) {
        ESP_LOGE(TAG, "esp_nimble_hci_and_controller_deinit() failed with
↳error: %d", ret);
    }
}
```

If `nimble_port_freertos_init()` is used during initialization, then `nimble_port_freertos_deinit()` should be called in the host task after `nimble_port_run()`.

Return

- ESP_OK if the deinitialization is successful
- Appropriate error codes from `esp_err_t` in case of an error

Macros

BLE_HCI_UART_H4_NONE

BLE_HCI_UART_H4_CMD

BLE_HCI_UART_H4_ACL

BLE_HCI_UART_H4_SCO

BLE_HCI_UART_H4_EVT

2.1.6 ESP-BLE-MESH

With various features of ESP-BLE-MESH, users can create a managed flooding mesh network for several scenarios, such as lighting, sensor and etc.

For an ESP32 to join and work on a ESP-BLE-MESH network, it must be provisioned firstly. By provisioning, the ESP32, as an unprovisioned device, will join the ESP-BLE-MESH network and become a ESP-BLE-MESH node, communicating with other nodes within or beyond the radio range.

Apart from ESP-BLE-MESH nodes, inside ESP-BLE-MESH network, there is also ESP32 that works as ESP-BLE-MESH Provisioner, which could provision unprovisioned devices into ESP-BLE-MESH nodes and configure the nodes with various features.

For information how to start using ESP32 and ESP-BLE-MESH, please see the Section [Getting Started with ESP-BLE-MESH](#). If you are interested in information on ESP-BLE-MESH architecture, including some details of software implementation, please see Section [ESP-BLE-MESH Architecture](#).

Application Examples and Demos

Please refer to Sections [ESP-BLE-MESH Examples](#) and [ESP-BLE-MESH Demo Videos](#).

API Reference

ESP-BLE-MESH APIs are divided into the following parts:

- [ESP-BLE-MESH Definitions](#)
- [ESP-BLE-MESH Core API Reference](#)
- [ESP-BLE-MESH Models API Reference](#)

ESP-BLE-MESH Definitions

This section contains only one header file, which lists the following items of ESP-BLE-MESH.

- ID of all the models and related message opcodes
- Structs of model, element and Composition Data
- Structs of used by ESP-BLE-MESH Node/Provisioner for provisioning
- Structs used to transmit/receive messages
- Event types and related event parameters

Header File

- `bt/esp_ble_mesh/api/esp_ble_mesh_defs.h`

Unions

```
union esp_ble_mesh_prov_cb_param_t
#include <esp_ble_mesh_defs.h> BLE Mesh Node/Provisioner callback parameters union.
```

Public Members

```
struct esp_ble_mesh_prov_cb_param_t::ble_mesh_prov_register_comp_param prov_register_comp
    Event parameter of ESP_BLE_MESH_PROV_REGISTER_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_set_unprov_dev_name_comp_param node_set_unprov_dev_name
    Event parameter of ESP_BLE_MESH_NODE_SET_UNPROV_DEV_NAME_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_prov_enable_comp_param node_prov_enable_comp
    Event parameter of ESP_BLE_MESH_NODE_PROV_ENABLE_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_prov_disable_comp_param node_prov_disable_comp
    Event parameter of ESP_BLE_MESH_NODE_PROV_DISABLE_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_link_open_evt_param node_prov_link_open
    Event parameter of ESP_BLE_MESH_NODE_PROV_LINK_OPEN_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_link_close_evt_param node_prov_link_close
    Event parameter of ESP_BLE_MESH_NODE_PROV_LINK_CLOSE_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_output_num_evt_param node_prov_output_num
    Event parameter of ESP_BLE_MESH_NODE_PROV_OUTPUT_NUMBER_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_output_str_evt_param node_prov_output_str
    Event parameter of ESP_BLE_MESH_NODE_PROV_OUTPUT_STRING_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_input_evt_param node_prov_input
    Event parameter of ESP_BLE_MESH_NODE_PROV_INPUT_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_provision_complete_evt_param node_prov_complete
    Event parameter of ESP_BLE_MESH_NODE_PROV_COMPLETE_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_provision_reset_param node_prov_reset
    Event parameter of ESP_BLE_MESH_NODE_PROV_RESET_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_set_oob_pub_key_comp_param node_prov_set_oob_pub_key_c
    Event parameter of ESP_BLE_MESH_NODE_PROV_SET_OOB_PUB_KEY_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_input_number_comp_param node_prov_input_num_comp
    Event parameter of ESP_BLE_MESH_NODE_PROV_INPUT_NUM_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_input_string_comp_param node_prov_input_str_comp
    Event parameter of ESP_BLE_MESH_NODE_PROV_INPUT_STR_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_identity_enable_comp_param node_proxy_identity_enab
    Event parameter of ESP_BLE_MESH_NODE_PROXY_IDENTITY_ENABLE_COMP_EVT
```

```

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_gatt_enable_comp_param node_proxy_gatt_enable_comp
    Event parameter of ESP_BLE_MESH_NODE_PROXY_GATT_ENABLE_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_gatt_disable_comp_param node_proxy_gatt_disable_comp
    Event parameter of ESP_BLE_MESH_NODE_PROXY_GATT_DISABLE_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_node_add_local_net_key_comp_param node_add_net_key_comp
    Event parameter of ESP_BLE_MESH_NODE_ADD_LOCAL_NET_KEY_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_node_add_local_app_key_comp_param node_add_app_key_comp
    Event parameter of ESP_BLE_MESH_NODE_ADD_LOCAL_APP_KEY_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_node_bind_local_mod_app_comp_param node_bind_app_key_to
    Event parameter of ESP_BLE_MESH_NODE_BIND_APP_KEY_TO_MODEL_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_rcv_unprov_adv_pkt_param provisioner_rcv_unprov
    Event parameter of ESP_BLE_MESH_PROVISIONER_RECV_UNPROV_ADV_PKT_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_prov_enable_comp_param provisioner_prov_enable
    Event parameter of ESP_BLE_MESH_PROVISIONER_PROV_ENABLE_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_prov_disable_comp_param provisioner_prov_disable
    Event parameter of ESP_BLE_MESH_PROVISIONER_PROV_DISABLE_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_link_open_evt_param provisioner_prov_link_open
    Event parameter of ESP_BLE_MESH_PROVISIONER_PROV_LINK_OPEN_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_prov_read_oob_pub_key_evt_param provisioner_prov_read_oob
    Event parameter of ESP_BLE_MESH_PROVISIONER_PROV_READ_OOB_PUB_KEY_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_prov_input_evt_param provisioner_prov_input
    Event parameter of ESP_BLE_MESH_PROVISIONER_PROV_INPUT_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_prov_output_evt_param provisioner_prov_output
    Event parameter of ESP_BLE_MESH_PROVISIONER_PROV_OUTPUT_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_link_close_evt_param provisioner_prov_link_close
    Event parameter of ESP_BLE_MESH_PROVISIONER_PROV_LINK_CLOSE_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_prov_comp_param provisioner_prov_complete
    Event parameter of ESP_BLE_MESH_PROVISIONER_PROV_COMPLETE_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_add_unprov_dev_comp_param provisioner_add_unprov_dev
    Event parameter of ESP_BLE_MESH_PROVISIONER_ADD_UNPROV_DEV_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_prov_dev_with_addr_comp_param provisioner_prov_dev_with_addr
    Event parameter of ESP_BLE_MESH_PROVISIONER_PROV_DEV_WITH_ADDR_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_delete_dev_comp_param provisioner_delete_dev
    Event parameter of ESP_BLE_MESH_PROVISIONER_DELETE_DEV_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_set_dev_uuid_match_comp_param provisioner_set_dev_uuid
    Event parameter of ESP_BLE_MESH_PROVISIONER_SET_DEV_UUID_MATCH_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_set_prov_data_info_comp_param provisioner_set_prov_data_info
    Event parameter of ESP_BLE_MESH_PROVISIONER_SET_PROV_DATA_INFO_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_set_static_oob_val_comp_param provisioner_set_static_oob_val
    Event parameter of ESP_BLE_MESH_PROVISIONER_SET_STATIC_OOB_VALUE_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_set_primary_elem_addr_comp_param provisioner_set_primary_elem_addr
    Event parameter of ESP_BLE_MESH_PROVISIONER_SET_PRIMARY_ELEM_ADDR_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_prov_read_oob_pub_key_comp_param provisioner_prov_read_oob_pub_key_comp
    Event parameter of ESP_BLE_MESH_PROVISIONER_PROV_READ_OOB_PUB_KEY_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_prov_input_num_comp_param provisioner_prov_input_num
    Event parameter of ESP_BLE_MESH_PROVISIONER_PROV_INPUT_NUMBER_COMP_EVT

```

struct *esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_prov_input_str_comp_param* **provisioner_prov_inpu**
Event parameter of ESP_BLE_MESH_PROVISIONER_PROV_INPUT_STRING_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_set_node_name_comp_param* **provisioner_set_node**
Event parameter of ESP_BLE_MESH_PROVISIONER_SET_NODE_NAME_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_add_local_app_key_comp_param* **provisioner_add_ap**
Event parameter of ESP_BLE_MESH_PROVISIONER_ADD_LOCAL_APP_KEY_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_update_local_app_key_comp_param* **provisioner_upda**
Event parameter of ESP_BLE_MESH_PROVISIONER_UPDATE_LOCAL_APP_KEY_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_bind_local_mod_app_comp_param* **provisioner_bind**
Event parameter of ESP_BLE_MESH_PROVISIONER_BIND_APP_KEY_TO_MODEL_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_add_local_net_key_comp_param* **provisioner_add_ne**
Event parameter of ESP_BLE_MESH_PROVISIONER_ADD_LOCAL_NET_KEY_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_update_local_net_key_comp_param* **provisioner_updat**
Event parameter of ESP_BLE_MESH_PROVISIONER_UPDATE_LOCAL_NET_KEY_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_store_node_comp_data_comp_param* **provisioner_sto**
Event parameter of ESP_BLE_MESH_PROVISIONER_STORE_NODE_COMP_DATA_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_delete_node_with_uuid_comp_param* **provisioner_del**
Event parameter of ESP_BLE_MESH_PROVISIONER_DELETE_NODE_WITH_UUID_COMP_EVT

struct *esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_delete_node_with_addr_comp_param* **provisioner_del**
Event parameter of ESP_BLE_MESH_PROVISIONER_DELETE_NODE_WITH_ADDR_COMP_EVT

int **err_code**

Indicate the result of enabling/disabling to receive heartbeat messages by the Provisioner

Indicate the result of setting the heartbeat filter type by the Provisioner

Indicate the result of setting the heartbeat filter address by the Provisioner

Indicate the result of directly erasing settings by the Provisioner

Indicate the result of opening settings with index by the Provisioner

Indicate the result of opening settings with user id by the Provisioner

Indicate the result of closing settings with index by the Provisioner

Indicate the result of closing settings with user id by the Provisioner

Indicate the result of deleting settings with index by the Provisioner

Indicate the result of deleting settings with user id by the Provisioner

bool **enable**

Indicate enabling or disabling receiving heartbeat messages

struct *esp_ble_mesh_prov_cb_param_t::[anonymous]* **provisioner_enable_heartbeat_rcv_comp**
ESP_BLE_MESH_PROVISIONER_ENABLE_HEARTBEAT_RECV_COMP_EVT.

Event parameters of ESP_BLE_MESH_PROVISIONER_ENABLE_HEARTBEAT_RECV_COMP_EVT

uint8_t **type**

Type of the filter used for receiving heartbeat messages

struct *esp_ble_mesh_prov_cb_param_t::[anonymous]* **provisioner_set_heartbeat_filter_type_comp**
ESP_BLE_MESH_PROVISIONER_SET_HEARTBEAT_FILTER_TYPE_COMP_EVT.

Event parameters of ESP_BLE_MESH_PROVISIONER_SET_HEARTBEAT_FILTER_TYPE_COMP_EVT

uint8_t **op**

Operation (add, remove, clean)

uint16_t **hb_src**

Heartbeat source address

uint16_t hb_dst
Heartbeat destination address

struct esp_ble_mesh_prov_cb_param_t::[anonymous] provisioner_set_heartbeat_filter_info_comp
ESP_BLE_MESH_PROVISIONER_SET_HEARTBEAT_FILTER_INFO_COMP_EVT.
Event parameters of ESP_BLE_MESH_PROVISIONER_SET_HEARTBEAT_FILTER_INFO_COMP_EVT

uint8_t init_ttl
Heartbeat InitTTL

uint8_t rx_ttl
Heartbeat RxTTL

uint8_t hops
Heartbeat hops (InitTTL - RxTTL + 1)

uint16_t feature
Bit field of currently active features of the node

int8_t rssi
RSSI of the heartbeat message

struct esp_ble_mesh_prov_cb_param_t::[anonymous] provisioner_rcv_heartbeat
ESP_BLE_MESH_PROVISIONER_RECV_HEARTBEAT_MESSAGE_EVT.
Event parameters of ESP_BLE_MESH_PROVISIONER_RECV_HEARTBEAT_MESSAGE_EVT

struct esp_ble_mesh_prov_cb_param_t::[anonymous] provisioner_direct_erase_settings_comp
ESP_BLE_MESH_PROVISIONER_DRIECT_ERASE_SETTINGS_COMP_EVT.
Event parameters of ESP_BLE_MESH_PROVISIONER_DRIECT_ERASE_SETTINGS_COMP_EVT

uint8_t index
Index of Provisioner settings

struct esp_ble_mesh_prov_cb_param_t::[anonymous] provisioner_open_settings_with_index_comp
ESP_BLE_MESH_PROVISIONER_OPEN_SETTINGS_WITH_INDEX_COMP_EVT.
Event parameter of ESP_BLE_MESH_PROVISIONER_OPEN_SETTINGS_WITH_INDEX_COMP_EVT

char uid[ESP_BLE_MESH_SETTINGS_UID_SIZE + 1]
Provisioner settings user id

struct esp_ble_mesh_prov_cb_param_t::[anonymous] provisioner_open_settings_with_uid_comp
ESP_BLE_MESH_PROVISIONER_OPEN_SETTINGS_WITH_UID_COMP_EVT.
Event parameters of ESP_BLE_MESH_PROVISIONER_OPEN_SETTINGS_WITH_UID_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::[anonymous] provisioner_close_settings_with_index_comp
ESP_BLE_MESH_PROVISIONER_CLOSE_SETTINGS_WITH_INDEX_COMP_EVT.
Event parameter of ESP_BLE_MESH_PROVISIONER_CLOSE_SETTINGS_WITH_INDEX_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::[anonymous] provisioner_close_settings_with_uid_comp
ESP_BLE_MESH_PROVISIONER_CLOSE_SETTINGS_WITH_UID_COMP_EVT.
Event parameters of ESP_BLE_MESH_PROVISIONER_CLOSE_SETTINGS_WITH_UID_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::[anonymous] provisioner_delete_settings_with_index_comp
ESP_BLE_MESH_PROVISIONER_DELETE_SETTINGS_WITH_INDEX_COMP_EVT.
Event parameter of ESP_BLE_MESH_PROVISIONER_DELETE_SETTINGS_WITH_INDEX_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::[anonymous] provisioner_delete_settings_with_uid_comp
ESP_BLE_MESH_PROVISIONER_DELETE_SETTINGS_WITH_UID_COMP_EVT.
Event parameters of ESP_BLE_MESH_PROVISIONER_DELETE_SETTINGS_WITH_UID_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_set_fast_prov_info_comp_param set_fast_prov_info_comp
Event parameter of ESP_BLE_MESH_SET_FAST_PROV_INFO_COMP_EVT


```

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_set_fast_prov_action_comp_param set_fast_prov_action_comp
    Event parameter of ESP_BLE_MESH_SET_FAST_PROV_ACTION_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_heartbeat_msg_rcv_param heartbeat_msg_rcv
    Event parameter of ESP_BLE_MESH_HEARTBEAT_MESSAGE_RECV_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_lpn_enable_comp_param lpn_enable_comp
    Event parameter of ESP_BLE_MESH_LPN_ENABLE_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_lpn_disable_comp_param lpn_disable_comp
    Event parameter of ESP_BLE_MESH_LPN_DISABLE_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_lpn_poll_comp_param lpn_poll_comp
    Event parameter of ESP_BLE_MESH_LPN_POLL_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_lpn_friendship_establish_param lpn_friendship_establish
    Event parameter of ESP_BLE_MESH_LPN_FRIENDSHIP_ESTABLISH_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_lpn_friendship_terminate_param lpn_friendship_terminate
    Event parameter of ESP_BLE_MESH_LPN_FRIENDSHIP_TERMINATE_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_friend_friendship_establish_param frnd_friendship_establish
    Event parameter of ESP_BLE_MESH_FRIEND_FRIENDSHIP_ESTABLISH_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_friend_friendship_terminate_param frnd_friendship_terminate
    Event parameter of ESP_BLE_MESH_FRIEND_FRIENDSHIP_TERMINATE_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_client_rcv_adv_pkt_param proxy_client_rcv_adv_pkt
    Event parameter of ESP_BLE_MESH_PROXY_CLIENT_RECV_ADV_PKT_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_client_connected_param proxy_client_connected
    Event parameter of ESP_BLE_MESH_PROXY_CLIENT_CONNECTED_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_client_disconnected_param proxy_client_disconnected
    Event parameter of ESP_BLE_MESH_PROXY_CLIENT_DISCONNECTED_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_client_rcv_filter_status_param proxy_client_rcv_filt
    Event parameter of ESP_BLE_MESH_PROXY_CLIENT_RECV_FILTER_STATUS_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_client_connect_comp_param proxy_client_connect_comp
    Event parameter of ESP_BLE_MESH_PROXY_CLIENT_CONNECT_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_client_disconnect_comp_param proxy_client_disconnect
    Event parameter of ESP_BLE_MESH_PROXY_CLIENT_DISCONNECT_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_client_set_filter_type_comp_param proxy_client_set_filt
    Event parameter of ESP_BLE_MESH_PROXY_CLIENT_SET_FILTER_TYPE_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_client_add_filter_addr_comp_param proxy_client_add_fi
    Event parameter of ESP_BLE_MESH_PROXY_CLIENT_ADD_FILTER_ADDR_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_proxy_client_remove_filter_addr_comp_param proxy_client_remo
    Event parameter of ESP_BLE_MESH_PROXY_CLIENT_REMOVE_FILTER_ADDR_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_model_sub_group_addr_comp_param model_sub_group_addr_co
    Event parameters of ESP_BLE_MESH_MODEL_SUBSCRIBE_GROUP_ADDR_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_model_unsub_group_addr_comp_param model_unsub_group_add
    Event parameters of ESP_BLE_MESH_MODEL_UNSUBSCRIBE_GROUP_ADDR_COMP_EVT

struct esp_ble_mesh_prov_cb_param_t::ble_mesh_deinit_mesh_comp_param deinit_mesh_comp
    Event parameter of ESP_BLE_MESH_DEINIT_MESH_COMP_EVT

struct ble_mesh_deinit_mesh_comp_param
    #include <esp_ble_mesh_defs.h> ESP_BLE_MESH_DEINIT_MESH_COMP_EVT.

```

Public Members

int err_code
Indicate the result of BLE Mesh deinitialization

struct ble_mesh_friend_friendship_establish_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_FRIEND_FRIENDSHIP_ESTABLISH_EVT.

Public Members

uint16_t lpn_addr
Low Power Node unicast address

struct ble_mesh_friend_friendship_terminate_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_FRIEND_FRIENDSHIP_TERMINATE_EVT.

Public Types

enum [anonymous]
This enum value is the reason of friendship termination on the friend node side

Values:

ESP_BLE_MESH_FRND_FRIENDSHIP_TERMINATE_ESTABLISH_FAIL
Friend Offer has been sent, but Friend Offer is not received within 1 second, friendship fails to be established

ESP_BLE_MESH_FRND_FRIENDSHIP_TERMINATE_POLL_TIMEOUT
Friendship is established, PollTimeout timer expires and no Friend Poll/Sub Add/Sub Remove is received

ESP_BLE_MESH_FRND_FRIENDSHIP_TERMINATE_RECV_FRND_REQ
Receive Friend Request from existing Low Power Node

ESP_BLE_MESH_FRND_FRIENDSHIP_TERMINATE_RECV_FRND_CLEAR
Receive Friend Clear from other friend node

ESP_BLE_MESH_FRND_FRIENDSHIP_TERMINATE_DISABLE
Friend feature disabled or corresponding NetKey is deleted

Public Members

uint16_t lpn_addr
Low Power Node unicast address

*esp_ble_mesh_prov_cb_param_t::ble_mesh_friend_friendship_terminate_param::***[anonymous] reason**
This enum value is the reason of friendship termination on the friend node side Friendship terminated reason

struct ble_mesh_heartbeat_msg_rcv_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_HEARTBEAT_MESSAGE_RECV_EVT.

Public Members

uint8_t hops
Heartbeat hops (InitTTL - RxTTL + 1)

uint16_t feature
Bit field of currently active features of the node


```
struct ble_mesh_input_evt_param  
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_PROV_INPUT_EVT.
```

Public Members

esp_ble_mesh_input_action_t **action**
Action of Input OOB Authentication

uint8_t size
Size of Input OOB Authentication

```
struct ble_mesh_input_number_comp_param  
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_PROV_INPUT_NUM_COMP_EVT.
```

Public Members

int err_code
Indicate the result of inputting number

```
struct ble_mesh_input_string_comp_param  
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_PROV_INPUT_STR_COMP_EVT.
```

Public Members

int err_code
Indicate the result of inputting string

```
struct ble_mesh_link_close_evt_param  
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_PROV_LINK_CLOSE_EVT.
```

Public Members

esp_ble_mesh_prov_bearer_t **bearer**
Type of the bearer used when device link is closed

```
struct ble_mesh_link_open_evt_param  
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_PROV_LINK_OPEN_EVT.
```

Public Members

esp_ble_mesh_prov_bearer_t **bearer**
Type of the bearer used when device link is open

```
struct ble_mesh_lpn_disable_comp_param  
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_LPN_DISABLE_COMP_EVT.
```

Public Members

int err_code
Indicate the result of disabling LPN functionality

```
struct ble_mesh_lpn_enable_comp_param  
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_LPN_ENABLE_COMP_EVT.
```

Public Members

int **err_code**
Indicate the result of enabling LPN functionality

struct ble_mesh_lpn_friendship_establish_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_LPN_FRIENDSHIP_ESTABLISH_EVT.

Public Members

uint16_t **friend_addr**
Friend Node unicast address

struct ble_mesh_lpn_friendship_terminate_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_LPN_FRIENDSHIP_TERMINATE_EVT.

Public Members

uint16_t **friend_addr**
Friend Node unicast address

struct ble_mesh_lpn_poll_comp_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_LPN_POLL_COMP_EVT.

Public Members

int **err_code**
Indicate the result of sending Friend Poll

struct ble_mesh_model_sub_group_addr_comp_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_MODEL_SUBSCRIBE_GROUP_ADDR_COMP_EVT.

Public Members

int **err_code**
Indicate the result of local model subscribing group address

uint16_t **element_addr**
Element address

uint16_t **company_id**
Company ID

uint16_t **model_id**
Model ID

uint16_t **group_addr**
Group Address

struct ble_mesh_model_unsub_group_addr_comp_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_MODEL_UNSUBSCRIBE_GROUP_ADDR_COMP_EVT.

Public Members

int **err_code**
Indicate the result of local model unsubscribing group address

uint16_t **element_addr**

Element address

uint16_t **company_id**

Company ID

uint16_t **model_id**

Model ID

uint16_t **group_addr**

Group Address

struct ble_mesh_node_add_local_app_key_comp_param

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_ADD_LOCAL_APP_KEY_COMP_EVT.

Public Members

int **err_code**

Indicate the result of adding local AppKey by the node

uint16_t **net_idx**

NetKey Index

uint16_t **app_idx**

AppKey Index

struct ble_mesh_node_add_local_net_key_comp_param

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_ADD_LOCAL_NET_KEY_COMP_EVT.

Public Members

int **err_code**

Indicate the result of adding local NetKey by the node

uint16_t **net_idx**

NetKey Index

struct ble_mesh_node_bind_local_mod_app_comp_param

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_BIND_APP_KEY_TO_MODEL_COMP_EVT.

Public Members

int **err_code**

Indicate the result of binding AppKey with model by the node

uint16_t **element_addr**

Element address

uint16_t **app_idx**

AppKey Index

uint16_t **company_id**

Company ID

uint16_t **model_id**

Model ID

struct ble_mesh_output_num_evt_param

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_PROV_OUTPUT_NUMBER_EVT.

Public Members

esp_ble_mesh_output_action_t **action**
Action of Output OOB Authentication

uint32_t **number**
Number of Output OOB Authentication

struct ble_mesh_output_str_evt_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_PROV_OUTPUT_STRING_EVT.

Public Members

char **string**[8]
String of Output OOB Authentication

struct ble_mesh_prov_disable_comp_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_PROV_DISABLE_COMP_EVT.

Public Members

int **err_code**
Indicate the result of disabling BLE Mesh device

struct ble_mesh_prov_enable_comp_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_PROV_ENABLE_COMP_EVT.

Public Members

int **err_code**
Indicate the result of enabling BLE Mesh device

struct ble_mesh_prov_register_comp_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROV_REGISTER_COMP_EVT.

Public Members

int **err_code**
Indicate the result of BLE Mesh initialization

struct ble_mesh_provision_complete_evt_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_PROV_COMPLETE_EVT.

Public Members

uint16_t **net_idx**
NetKey Index

uint8_t **net_key**[16]
NetKey

uint16_t **addr**
Primary address

uint8_t **flags**
Flags

`uint32_t iv_index`
IV Index

struct ble_mesh_provisioner_reset_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_PROV_RESET_EVT.

struct ble_mesh_provisioner_add_local_app_key_comp_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_ADD_LOCAL_APP_KEY_COMP_EVT.

Public Members

`int err_code`
Indicate the result of adding local AppKey by the Provisioner

`uint16_t net_idx`
NetKey Index

`uint16_t app_idx`
AppKey Index

struct ble_mesh_provisioner_add_local_net_key_comp_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_ADD_LOCAL_NET_KEY_COMP_EVT.

Public Members

`int err_code`
Indicate the result of adding local NetKey by the Provisioner

`uint16_t net_idx`
NetKey Index

struct ble_mesh_provisioner_add_unprov_dev_comp_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_ADD_UNPROV_DEV_COMP_EVT.

Public Members

`int err_code`
Indicate the result of adding device into queue by the Provisioner

struct ble_mesh_provisioner_bind_local_mod_app_comp_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_BIND_APP_KEY_TO_MODEL_COMP_EVT.

Public Members

`int err_code`
Indicate the result of binding AppKey with model by the Provisioner

`uint16_t element_addr`
Element address

`uint16_t app_idx`
AppKey Index

`uint16_t company_id`
Company ID

`uint16_t model_id`
Model ID

struct ble_mesh_provisioner_delete_dev_comp_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_DELETE_DEV_COMP_EVT.

Public Members

int **err_code**
Indicate the result of deleting device by the Provisioner

struct ble_mesh_provisioner_delete_node_with_addr_comp_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_DELETE_NODE_WITH_ADDR_COMP_EVT.

Public Members

int **err_code**
Indicate the result of deleting node with unicast address by the Provisioner

uint16_t **unicast_addr**
Node unicast address

struct ble_mesh_provisioner_delete_node_with_uuid_comp_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_DELETE_NODE_WITH_UUID_COMP_EVT.

Public Members

int **err_code**
Indicate the result of deleting node with uuid by the Provisioner

uint8_t **uuid**[16]
Node device uuid

struct ble_mesh_provisioner_link_close_evt_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_PROV_LINK_CLOSE_EVT.

Public Members

esp_ble_mesh_prov_bearer_t **bearer**
Type of the bearer used when Provisioner link is closed

uint8_t **reason**
Reason of the closed provisioning link

struct ble_mesh_provisioner_link_open_evt_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_PROV_LINK_OPEN_EVT.

Public Members

esp_ble_mesh_prov_bearer_t **bearer**
Type of the bearer used when Provisioner link is opened

struct ble_mesh_provisioner_prov_comp_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_PROV_COMPLETE_EVT.

Public Members

uint16_t **node_idx**
Index of the provisioned device

esp_ble_mesh_octet16_t **device_uuid**
Device UUID of the provisioned device

`uint16_t unicast_addr`
Primary address of the provisioned device

`uint8_t element_num`
Element count of the provisioned device

`uint16_t netkey_idx`
NetKey Index of the provisioned device

struct ble_mesh_provisioner_prov_dev_with_addr_comp_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_PROV_DEV_WITH_ADDR_COMP_EVT.

Public Members

`int err_code`
Indicate the result of Provisioner starting to provision a device

struct ble_mesh_provisioner_prov_disable_comp_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_PROV_DISABLE_COMP_EVT.

Public Members

`int err_code`
Indicate the result of disabling BLE Mesh Provisioner

struct ble_mesh_provisioner_prov_enable_comp_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_PROV_ENABLE_COMP_EVT.

Public Members

`int err_code`
Indicate the result of enabling BLE Mesh Provisioner

struct ble_mesh_provisioner_prov_input_evt_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_PROV_INPUT_EVT.

Public Members

esp_ble_mesh_oob_method_t **method**
Method of device Output OOB Authentication

esp_ble_mesh_output_action_t **action**
Action of device Output OOB Authentication

`uint8_t size`
Size of device Output OOB Authentication

`uint8_t link_idx`
Index of the provisioning link

struct ble_mesh_provisioner_prov_input_num_comp_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_PROV_INPUT_NUMBER_COMP_EVT.

Public Members

`int err_code`
Indicate the result of inputting number by the Provisioner

```
struct ble_mesh_provisioner_prov_input_str_comp_param  
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_PROV_INPUT_STRING_COMP_EVT.
```

Public Members

int err_code
Indicate the result of inputting string by the Provisioner

```
struct ble_mesh_provisioner_prov_output_evt_param  
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_PROV_OUTPUT_EVT.
```

Public Members

esp_ble_mesh_oob_method_t **method**
Method of device Input OOB Authentication

esp_ble_mesh_input_action_t **action**
Action of device Input OOB Authentication

uint8_t size
Size of device Input OOB Authentication

uint8_t link_idx
Index of the provisioning link

char string[8]
String output by the Provisioner

uint32_t number
Number output by the Provisioner

union esp_ble_mesh_prov_cb_param_t::ble_mesh_provisioner_prov_output_evt_param::[anonymous] [anonymous]

```
struct ble_mesh_provisioner_prov_read_oob_pub_key_comp_param  
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_PROV_READ_OOB_PUB_KEY_COMP_EVT.
```

Public Members

int err_code
Indicate the result of setting OOB Public Key by the Provisioner

```
struct ble_mesh_provisioner_prov_read_oob_pub_key_evt_param  
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_PROV_READ_OOB_PUB_KEY_EVT.
```

Public Members

uint8_t link_idx
Index of the provisioning link

```
struct ble_mesh_provisioner_rcv_unprov_adv_pkt_param  
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_RECV_UNPROV_ADV_PKT_EVT.
```

Public Members

uint8_t dev_uuid[16]
Device UUID of the unprovisioned device

esp_ble_mesh_bd_addr_t **addr**
Device address of the unprovisioned device

esp_ble_mesh_addr_type_t **addr_type**

Device address type

uint16_t **oob_info**

OOB Info of the unprovisioned device

uint8_t **adv_type**

Advertising type of the unprovisioned device

esp_ble_mesh_prov_bearer_t **bearer**

Bearer of the unprovisioned device

int8_t **rsssi**

RSSI of the received advertising packet

struct ble_mesh_provisioner_set_dev_uuid_match_comp_param

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_SET_DEV_UUID_MATCH_COMP_EVT.

Public Members

int **err_code**

Indicate the result of setting Device UUID match value by the Provisioner

struct ble_mesh_provisioner_set_node_name_comp_param

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_SET_NODE_NAME_COMP_EVT.

Public Members

int **err_code**

Indicate the result of setting provisioned device name by the Provisioner

uint16_t **node_index**

Index of the provisioned device

struct ble_mesh_provisioner_set_primary_elem_addr_comp_param

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_SET_PRIMARY_ELEM_ADDR_COMP_EVT.

Public Members

int **err_code**

Indicate the result of setting unicast address of primary element by the Provisioner

struct ble_mesh_provisioner_set_prov_data_info_comp_param

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_SET_PROV_DATA_INFO_COMP_EVT.

Public Members

int **err_code**

Indicate the result of setting provisioning info by the Provisioner

struct ble_mesh_provisioner_set_static_oob_val_comp_param

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_SET_STATIC_OOB_VALUE_COMP_EVT.

Public Members

int **err_code**

Indicate the result of setting static oob value by the Provisioner

```
struct ble_mesh_provisioner_store_node_comp_data_comp_param  
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_STORE_NODE_COMP_DATA_COMP_EVT.
```

Public Members

```
int err_code  
    Indicate the result of storing node composition data by the Provisioner  
  
uint16_t addr  
    Node element address
```

```
struct ble_mesh_provisioner_update_local_app_key_comp_param  
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_UPDATE_LOCAL_APP_KEY_COMP_EVT.
```

Public Members

```
int err_code  
    Indicate the result of updating local AppKey by the Provisioner  
  
uint16_t net_idx  
    NetKey Index  
  
uint16_t app_idx  
    AppKey Index
```

```
struct ble_mesh_provisioner_update_local_net_key_comp_param  
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROVISIONER_UPDATE_LOCAL_NET_KEY_COMP_EVT.
```

Public Members

```
int err_code  
    Indicate the result of updating local NetKey by the Provisioner  
  
uint16_t net_idx  
    NetKey Index
```

```
struct ble_mesh_proxy_client_add_filter_addr_comp_param  
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROXY_CLIENT_ADD_FILTER_ADDR_COMP_EVT.
```

Public Members

```
int err_code  
    Indicate the result of Proxy Client add filter address  
  
uint8_t conn_handle  
    Proxy connection handle  
  
uint16_t net_idx  
    Corresponding NetKey Index
```

```
struct ble_mesh_proxy_client_connect_comp_param  
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROXY_CLIENT_CONNECT_COMP_EVT.
```

Public Members

```
int err_code  
    Indicate the result of Proxy Client connect
```

esp_ble_mesh_bd_addr_t **addr**

Device address of the Proxy Server

esp_ble_mesh_addr_type_t **addr_type**

Device address type

uint16_t **net_idx**

Corresponding NetKey Index

struct ble_mesh_proxy_client_connected_param

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROXY_CLIENT_CONNECTED_EVT.

Public Members

esp_ble_mesh_bd_addr_t **addr**

Device address of the Proxy Server

esp_ble_mesh_addr_type_t **addr_type**

Device address type

uint8_t **conn_handle**

Proxy connection handle

uint16_t **net_idx**

Corresponding NetKey Index

struct ble_mesh_proxy_client_disconnect_comp_param

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROXY_CLIENT_DISCONNECT_COMP_EVT.

Public Members

int **err_code**

Indicate the result of Proxy Client disconnect

uint8_t **conn_handle**

Proxy connection handle

struct ble_mesh_proxy_client_disconnected_param

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROXY_CLIENT_DISCONNECTED_EVT.

Public Members

esp_ble_mesh_bd_addr_t **addr**

Device address of the Proxy Server

esp_ble_mesh_addr_type_t **addr_type**

Device address type

uint8_t **conn_handle**

Proxy connection handle

uint16_t **net_idx**

Corresponding NetKey Index

uint8_t **reason**

Proxy disconnect reason

struct ble_mesh_proxy_client_rcv_adv_pkt_param

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROXY_CLIENT_RECV_ADV_PKT_EVT.

Public Members

esp_ble_mesh_bd_addr_t **addr**

Device address

esp_ble_mesh_addr_type_t **addr_type**

Device address type

uint16_t **net_idx**

Network ID related NetKey Index

uint8_t **net_id**[8]

Network ID contained in the advertising packet

int8_t **rss**

RSSI of the received advertising packet

struct ble_mesh_proxy_client_rcv_filter_status_param

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROXY_CLIENT_RECV_FILTER_STATUS_EVT.

Public Members

uint8_t **conn_handle**

Proxy connection handle

uint16_t **server_addr**

Proxy Server primary element address

uint16_t **net_idx**

Corresponding NetKey Index

uint8_t **filter_type**

Proxy Server filter type(whitelist or blacklist)

uint16_t **list_size**

Number of addresses in the Proxy Server filter list

struct ble_mesh_proxy_client_remove_filter_addr_comp_param

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROXY_CLIENT_REMOVE_FILTER_ADDR_COMP_EVT.

Public Members

int **err_code**

Indicate the result of Proxy Client remove filter address

uint8_t **conn_handle**

Proxy connection handle

uint16_t **net_idx**

Corresponding NetKey Index

struct ble_mesh_proxy_client_set_filter_type_comp_param

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_PROXY_CLIENT_SET_FILTER_TYPE_COMP_EVT.

Public Members

int **err_code**

Indicate the result of Proxy Client set filter type

uint8_t **conn_handle**

Proxy connection handle

`uint16_t net_idx`
Corresponding NetKey Index

struct ble_mesh_proxy_gatt_disable_comp_param
#include <esp_ble_mesh_defs.h>ESP_BLE_MESH_NODE_PROXY_GATT_DISABLE_COMP_EVT.

Public Members

`int err_code`
Indicate the result of disabling Mesh Proxy Service

struct ble_mesh_proxy_gatt_enable_comp_param
#include <esp_ble_mesh_defs.h>ESP_BLE_MESH_NODE_PROXY_GATT_ENABLE_COMP_EVT.

Public Members

`int err_code`
Indicate the result of enabling Mesh Proxy Service

struct ble_mesh_proxy_identity_enable_comp_param
#include <esp_ble_mesh_defs.h>ESP_BLE_MESH_NODE_PROXY_IDENTITY_ENABLE_COMP_EVT.

Public Members

`int err_code`
Indicate the result of enabling Mesh Proxy advertising

struct ble_mesh_set_fast_prov_action_comp_param
#include <esp_ble_mesh_defs.h>ESP_BLE_MESH_SET_FAST_PROV_ACTION_COMP_EVT.

Public Members

`uint8_t status_action`
Indicate the result of setting action of fast provisioning

struct ble_mesh_set_fast_prov_info_comp_param
#include <esp_ble_mesh_defs.h>ESP_BLE_MESH_SET_FAST_PROV_INFO_COMP_EVT.

Public Members

`uint8_t status_unicast`
Indicate the result of setting unicast address range of fast provisioning

`uint8_t status_net_idx`
Indicate the result of setting NetKey Index of fast provisioning

`uint8_t status_match`
Indicate the result of setting matching Device UUID of fast provisioning

struct ble_mesh_set_oob_pub_key_comp_param
#include <esp_ble_mesh_defs.h>ESP_BLE_MESH_NODE_PROV_SET_OOB_PUB_KEY_COMP_EVT.

Public Members

int **err_code**
Indicate the result of setting OOB Public Key

struct ble_mesh_set_unprov_dev_name_comp_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_NODE_SET_UNPROV_DEV_NAME_COMP_EVT.

Public Members

int **err_code**
Indicate the result of setting BLE Mesh device name

union esp_ble_mesh_server_state_value_t
#include <esp_ble_mesh_defs.h> Server model state value union.

Public Members

uint8_t **onoff**
The value of the Generic OnOff state

The value of the Light LC Light OnOff state

struct esp_ble_mesh_server_state_value_t::[anonymous] gen_onoff
The Generic OnOff state

int16_t **level**
The value of the Generic Level state

struct esp_ble_mesh_server_state_value_t::[anonymous] gen_level
The Generic Level state

uint8_t **onpowerup**
The value of the Generic OnPowerUp state

struct esp_ble_mesh_server_state_value_t::[anonymous] gen_onpowerup
The Generic OnPowerUp state

uint16_t **power**
The value of the Generic Power Actual state

struct esp_ble_mesh_server_state_value_t::[anonymous] gen_power_actual
The Generic Power Actual state

uint16_t **lightness**
The value of the Light Lightness Actual state

The value of the Light Lightness Linear state

The value of the Light CTL Lightness state

The value of the Light HSL Lightness state

The value of the Light xyL Lightness state

struct esp_ble_mesh_server_state_value_t::[anonymous] light_lightness_actual
The Light Lightness Actual state

struct esp_ble_mesh_server_state_value_t::[anonymous] light_lightness_linear
The Light Lightness Linear state

struct esp_ble_mesh_server_state_value_t::[anonymous] light_ctl_lightness
The Light CTL Lightness state

uint16_t **temperature**
The value of the Light CTL Temperature state

int16_t delta_uv

The value of the Light CTL Delta UV state

struct esp_ble_mesh_server_state_value_t::[anonymous] light_ctl_temp_delta_uv

The Light CTL Temperature & Delta UV states

uint16_t hue

The value of the Light HSL Hue state

uint16_t saturation

The value of the Light HSL Saturation state

struct esp_ble_mesh_server_state_value_t::[anonymous] light_hsl

The Light HSL composite state

struct esp_ble_mesh_server_state_value_t::[anonymous] light_hsl_lightness

The Light HSL Lightness state

struct esp_ble_mesh_server_state_value_t::[anonymous] light_hsl_hue

The Light HSL Hue state

struct esp_ble_mesh_server_state_value_t::[anonymous] light_hsl_saturation

The Light HSL Saturation state

struct esp_ble_mesh_server_state_value_t::[anonymous] light_xyl_lightness

The Light xyL Lightness state

struct esp_ble_mesh_server_state_value_t::[anonymous] light_lc_light_onoff

The Light LC Light OnOff state

union esp_ble_mesh_model_cb_param_t

#include <esp_ble_mesh_defs.h> BLE Mesh model callback parameters union.

Public Members

struct esp_ble_mesh_model_cb_param_t::ble_mesh_model_operation_evt_param model_operation

Event parameter of ESP_BLE_MESH_MODEL_OPERATION_EVT

struct esp_ble_mesh_model_cb_param_t::ble_mesh_model_send_comp_param model_send_comp

Event parameter of ESP_BLE_MESH_MODEL_SEND_COMP_EVT

struct esp_ble_mesh_model_cb_param_t::ble_mesh_model_publish_comp_param model_publish_comp

Event parameter of ESP_BLE_MESH_MODEL_PUBLISH_COMP_EVT

struct esp_ble_mesh_model_cb_param_t::ble_mesh_mod_rcv_publish_msg_param client_rcv_publish_msg

Event parameter of ESP_BLE_MESH_CLIENT_MODEL_RECV_PUBLISH_MSG_EVT

struct esp_ble_mesh_model_cb_param_t::ble_mesh_client_model_send_timeout_param client_send_timeout

Event parameter of ESP_BLE_MESH_CLIENT_MODEL_SEND_TIMEOUT_EVT

struct esp_ble_mesh_model_cb_param_t::ble_mesh_model_publish_update_evt_param model_publish_update

Event parameter of ESP_BLE_MESH_MODEL_PUBLISH_UPDATE_EVT

struct esp_ble_mesh_model_cb_param_t::ble_mesh_server_model_update_state_comp_param server_model_update

Event parameter of ESP_BLE_MESH_SERVER_MODEL_UPDATE_STATE_COMP_EVT

struct ble_mesh_client_model_send_timeout_param

#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_CLIENT_MODEL_SEND_TIMEOUT_EVT.

Public Members

uint32_t opcode

Opcode of the previously sent message

esp_ble_mesh_model_t ***model**
Pointer to the model which sends the previous message

esp_ble_mesh_msg_ctx_t ***ctx**
Pointer to the context of the previous message

struct ble_mesh_mod_rcv_publish_msg_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_CLIENT_MODEL_RECV_PUBLISH_MSG_EVT.

Public Members

uint32_t **opcode**
Opcode of the unsolicited received message

esp_ble_mesh_model_t ***model**
Pointer to the model which receives the message

esp_ble_mesh_msg_ctx_t ***ctx**
Pointer to the context of the message

uint16_t **length**
Length of the received message

uint8_t ***msg**
Value of the received message

struct ble_mesh_model_operation_evt_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_MODEL_OPERATION_EVT.

Public Members

uint32_t **opcode**
Opcode of the received message

esp_ble_mesh_model_t ***model**
Pointer to the model which receives the message

esp_ble_mesh_msg_ctx_t ***ctx**
Pointer to the context of the received message

uint16_t **length**
Length of the received message

uint8_t ***msg**
Value of the received message

struct ble_mesh_model_publish_comp_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_MODEL_PUBLISH_COMP_EVT.

Public Members

int **err_code**
Indicate the result of publishing a message

esp_ble_mesh_model_t ***model**
Pointer to the model which publishes the message

struct ble_mesh_model_publish_update_evt_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_MODEL_PUBLISH_UPDATE_EVT.

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the model which is going to update its publish message

struct ble_mesh_model_send_comp_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_MODEL_SEND_COMP_EVT.

Public Members

int **err_code**

Indicate the result of sending a message

uint32_t **opcode**

Opcode of the message

esp_ble_mesh_model_t ***model**

Pointer to the model which sends the message

esp_ble_mesh_msg_ctx_t ***ctx**

Context of the message

struct ble_mesh_server_model_update_state_comp_param
#include <esp_ble_mesh_defs.h> ESP_BLE_MESH_SERVER_MODEL_UPDATE_STATE_COMP_EVT.

Public Members

int **err_code**

Indicate the result of updating server model state

esp_ble_mesh_model_t ***model**

Pointer to the server model which state value is updated

esp_ble_mesh_server_state_type_t **type**

Type of the updated server state

Structures

struct esp_ble_mesh_deinit_param_t
BLE Mesh deinit parameters

Public Members

bool **erase_flash**

Indicate if erasing flash when deinit mesh stack

struct esp_ble_mesh_elem_t

Abstraction that describes a BLE Mesh Element. This structure is associated with struct bt_mesh_elem in mesh_access.h

Public Members

uint16_t **element_addr**

Element Address, assigned during provisioning.

const uint16_t **location**

Location Descriptor (GATT Bluetooth Namespace Descriptors)

const uint8_t **sig_model_count**

SIG Model count

const uint8_t vnd_model_count

Vendor Model count

esp_ble_mesh_model_t *sig_models

SIG Models

esp_ble_mesh_model_t *vnd_models

Vendor Models

struct esp_ble_mesh_model_pub_t

Abstraction that describes a model publication context. This structure is associated with struct `bt_mesh_model_pub` in `mesh_access.h`

Public Members

esp_ble_mesh_model_t *model

Pointer to the model to which the context belongs. Initialized by the stack.

uint16_t **publish_addr**

Publish Address.

uint16_t **app_idx** : 12

Publish AppKey Index.

uint16_t **cred** : 1

Friendship Credentials Flag.

uint16_t **send_rel** : 1

Force reliable sending (segment acks)

uint8_t **tvl**

Publish Time to Live.

uint8_t **retransmit**

Retransmit Count & Interval Steps.

uint8_t **period**

Publish Period.

uint8_t **period_div** : 4

Divisor for the Period.

uint8_t **fast_period** : 1

Use FastPeriodDivisor

uint8_t **count** : 3

Retransmissions left.

uint32_t **period_start**

Start of the current period.

struct net_buf_simple *msg

Publication buffer, containing the publication message.

This will get correctly created when the publication context has been defined using the `ESP_BLE_MESH_MODEL_PUB_DEFINE` macro.

`ESP_BLE_MESH_MODEL_PUB_DEFINE(name, size);`

esp_ble_mesh_cb_t **update**

Callback used to update publish message. Initialized by the stack.

struct k_delayed_work timer

Publish Period Timer. Initialized by the stack.

uint8_t **dev_role**

Role of the device that is going to publish messages

struct esp_ble_mesh_model_op_t

Abstraction that describes a model operation context. This structure is associated with struct `bt_mesh_model_op` in `mesh_access.h`

Public Members

const uint32_t opcode

Message opcode

const size_t min_len

Message minimum length

esp_ble_mesh_cb_t **param_cb**

Callback used to handle message. Initialized by the stack.

struct esp_ble_mesh_model_cbs_t

Abstraction that describes a model callback structure. This structure is associated with struct `bt_mesh_model_cb` in `mesh_access.h`.

Public Members

esp_ble_mesh_cb_t **init_cb**

Callback used during model initialization. Initialized by the stack.

struct esp_ble_mesh_model

Abstraction that describes a Mesh Model instance. This structure is associated with struct `bt_mesh_model` in `mesh_access.h`

Public Members

union esp_ble_mesh_model::[anonymous] [anonymous]

Model ID

uint8_t element_idx

Internal information, mainly for persistent storage Belongs to Nth element

uint8_t model_idx

Is the Nth model in the element

uint16_t flags

Information about what has changed

esp_ble_mesh_elem_t ***element**

The Element to which this Model belongs

esp_ble_mesh_model_pub_t ***const pub**

Model Publication

uint16_t keys[CONFIG_BLE_MESH_MODEL_KEY_COUNT]

AppKey List

uint16_t groups[CONFIG_BLE_MESH_MODEL_GROUP_COUNT]

Subscription List (group or virtual addresses)

esp_ble_mesh_model_op_t ***op**

Model operation context

esp_ble_mesh_model_cbs_t ***cb**

Model callback structure

void *user_data

Model-specific user data

struct esp_ble_mesh_msg_ctx_t

Message sending context. This structure is associated with struct bt_mesh_msg_ctx in mesh_access.h

Public Members**uint16_t net_idx**

NetKey Index of the subnet through which to send the message.

uint16_t app_idx

AppKey Index for message encryption.

uint16_t addr

Remote address.

uint16_t recv_dst

Destination address of a received message. Not used for sending.

int8_t recv_rssi

RSSI of received packet. Not used for sending.

uint8_t recv_ttl : 7

Received TTL value. Not used for sending.

uint8_t send_rel : 1

Force sending reliably by using segment acknowledgement

uint8_t send_ttl

TTL, or ESP_BLE_MESH_TTL_DEFAULT for default TTL.

uint32_t recv_op

Opcode of a received message. Not used for sending message.

***esp_ble_mesh_model_t* *model**

Model corresponding to the message, no need to be initialized before sending message

bool srv_send

Indicate if the message is sent by a node server model, no need to be initialized before sending message

struct esp_ble_mesh_prov_t

Provisioning properties & capabilities. This structure is associated with struct bt_mesh_prov in mesh_access.h

struct esp_ble_mesh_comp_t

Node Composition data context. This structure is associated with struct bt_mesh_comp in mesh_access.h

Public Members**uint16_t cid**

16-bit SIG-assigned company identifier

uint16_t pid

16-bit vendor-assigned product identifier

uint16_t vid

16-bit vendor-assigned product version identifier

size_t element_count

Element count

***esp_ble_mesh_elem_t* *elements**

A sequence of elements

struct esp_ble_mesh_unprov_dev_add_t

Information of the device which is going to be added for provisioning.

Public Members

esp_ble_mesh_bd_addr_t **addr**

Device address

esp_ble_mesh_addr_type_t **addr_type**

Device address type

uint8_t **uuid**[16]

Device UUID

uint16_t **oob_info**

Device OOB Info ADD_DEV_START_PROV_NOW_FLAG shall not be set if the bearer has both PB-ADV and PB-GATT enabled

esp_ble_mesh_prov_bearer_t **bearer**

Provisioning Bearer

struct esp_ble_mesh_device_delete_t

Information of the device which is going to be deleted.

Public Members

esp_ble_mesh_bd_addr_t **addr**

Device address

esp_ble_mesh_addr_type_t **addr_type**

Device address type

uint8_t **uuid**[16]

Device UUID

uint8_t **flag**

BIT0: device address; BIT1: device UUID

struct esp_ble_mesh_prov_data_info_t

Information of the provisioner which is going to be updated.

Public Members

uint16_t **net_idx**

NetKey Index

uint8_t **flags**

Flags

uint32_t **iv_index**

IV Index

uint8_t **flag**

BIT0: net_idx; BIT1: flags; BIT2: iv_index

struct esp_ble_mesh_node_t

Information of the provisioned node

Public Members

esp_ble_mesh_bd_addr_t **addr**

Node device address

esp_ble_mesh_addr_type_t **addr_type**

Node device address type

`uint8_t dev_uuid[16]`
Device UUID

`uint16_t oob_info`
Node OOB information

`uint16_t unicast_addr`
Node unicast address

`uint8_t element_num`
Node element number

`uint16_t net_idx`
Node NetKey Index

`uint8_t flags`
Node key refresh flag and iv update flag

`uint32_t iv_index`
Node IV Index

`uint8_t dev_key[16]`
Node device key

`char name[ESP_BLE_MESH_NODE_NAME_MAX_LEN + 1]`
Node name

`uint16_t comp_length`
Length of Composition Data

`uint8_t *comp_data`
Value of Composition Data

struct esp_ble_mesh_fast_prov_info_t
Context of fast provisioning which need to be set.

Public Members

`uint16_t unicast_min`
Minimum unicast address used for fast provisioning

`uint16_t unicast_max`
Maximum unicast address used for fast provisioning

`uint16_t net_idx`
Netkey index used for fast provisioning

`uint8_t flags`
Flags used for fast provisioning

`uint32_t iv_index`
IV Index used for fast provisioning

`uint8_t offset`
Offset of the UUID to be compared

`uint8_t match_len`
Length of the UUID to be compared

`uint8_t match_val[16]`
Value of UUID to be compared

struct esp_ble_mesh_heartbeat_filter_info_t
Context of Provisioner heartbeat filter information to be set

Public Members

`uint16_t hb_src`
Heartbeat source address (unicast address)

`uint16_t hb_dst`
Heartbeat destination address (unicast address or group address)

struct esp_ble_mesh_client_op_pair_t
BLE Mesh client models related definitions.
Client model Get/Set message opcode and corresponding Status message opcode

Public Members

`uint32_t cli_op`
The client message opcode

`uint32_t status_op`
The server status opcode corresponding to the client message opcode

struct esp_ble_mesh_client_t
Client Model user data context.

Public Members

`esp_ble_mesh_model_t *model`
Pointer to the client model. Initialized by the stack.

`int op_pair_size`
Size of the op_pair

`const esp_ble_mesh_client_op_pair_t *op_pair`
Table containing get/set message opcode and corresponding status message opcode

`uint32_t publish_status`
Callback used to handle the received unsolicited message. Initialized by the stack.

`void *internal_data`
Pointer to the internal data of client model

`uint8_t msg_role`
Role of the device (Node/Provisioner) that is going to send messages

struct esp_ble_mesh_client_common_param_t
Common parameters of the messages sent by Client Model.

Public Members

`esp_ble_mesh_opcode_t opcode`
Message opcode

`esp_ble_mesh_model_t *model`
Pointer to the client model structure

`esp_ble_mesh_msg_ctx_t ctx`
The context used to send message

`int32_t msg_timeout`
Timeout value (ms) to get response to the sent message Note: if using default timeout value in menuconfig, make sure to set this value to 0

uint8_t msg_role

Role of the device - Node/Provisioner

struct esp_ble_mesh_state_transition_t

Parameters of the server model state transition

Public Functions

BLE_MESH_ATOMIC_DEFINE (flag, *ESP_BLE_MESH_SERVER_FLAG_MAX*)

Flag used to indicate if the transition timer has been started internally.

If the model which contains *esp_ble_mesh_state_transition_t* sets “set_auto_rsp” to *ESP_BLE_MESH_SERVER_RSP_BY_APP*, the handler of the timer shall be initialized by the users.

And users can use this flag to indicate whether the timer is started or not.

Public Members

bool just_started

Indicate if the state transition has just started

uint8_t trans_time

State transition time

uint8_t remain_time

Remaining time of state transition

uint8_t delay

Delay before starting state transition

uint32_t quo_tt

Duration of each divided transition step

uint32_t counter

Number of steps which the transition duration is divided

uint32_t total_duration

State transition total duration

int64_t start_timestamp

Time when the state transition is started

struct k_delayed_work timer

Timer used for state transition

struct esp_ble_mesh_last_msg_info_t

Parameters of the server model received last same set message.

Public Members

uint8_t tid

Transaction number of the last message

uint16_t src

Source address of the last message

uint16_t dst

Destination address of the last message

int64_t timestamp

Time when the last message is received

struct esp_ble_mesh_server_rsp_ctrl_t

Parameters of the Server Model response control

Public Members**uint8_t get_auto_rsp** : 1

BLE Mesh Server Response Option.

1. If `get_auto_rsp` is set to `ESP_BLE_MESH_SERVER_RSP_BY_APP`, then the response of Client Get messages need to be replied by the application;
2. If `get_auto_rsp` is set to `ESP_BLE_MESH_SERVER_AUTO_RSP`, then the response of Client Get messages will be replied by the server models;
3. If `set_auto_rsp` is set to `ESP_BLE_MESH_SERVER_RSP_BY_APP`, then the response of Client Set messages need to be replied by the application;
4. If `set_auto_rsp` is set to `ESP_BLE_MESH_SERVER_AUTO_RSP`, then the response of Client Set messages will be replied by the server models;
5. If `status_auto_rsp` is set to `ESP_BLE_MESH_SERVER_RSP_BY_APP`, then the response of Server Status messages need to be replied by the application;
6. If `status_auto_rsp` is set to `ESP_BLE_MESH_SERVER_AUTO_RSP`, then the response of Server Status messages will be replied by the server models;

uint8_t set_auto_rsp : 1

Response control for Client Set messages

uint8_t status_auto_rsp : 1

Response control for Server Status messages

Macros**ESP_BLE_HOST_STATUS_ENABLED****ESP_BLE_HOST_STATUS_CHECK** (status)

The maximum length of a BLE Mesh message, including Opcode, Payload and TransMIC

ESP_BLE_MESH_SDU_MAX_LEN

Length of a short Mesh MIC.

ESP_BLE_MESH_MIC_SHORT

Length of a long Mesh MIC.

ESP_BLE_MESH_MIC_LONG

The maximum length of a BLE Mesh provisioned node name

ESP_BLE_MESH_NODE_NAME_MAX_LEN

The maximum length of a BLE Mesh unprovisioned device name

ESP_BLE_MESH_DEVICE_NAME_MAX_LEN

The maximum length of settings user id

ESP_BLE_MESH_SETTINGS_UID_SIZE

Invalid settings index

ESP_BLE_MESH_INVALID_SETTINGS_IDX

Define the BLE Mesh octet 16 bytes size

ESP_BLE_MESH_OCTET16_LEN**ESP_BLE_MESH_OCTET8_LEN****ESP_BLE_MESH_CID_NVAL**

Special TTL value to request using configured default TTL

ESP_BLE_MESH_TTL_DEFAULT

Maximum allowed TTL value

ESP_BLE_MESH_TTL_MAX

ESP_BLE_MESH_ADDR_UNASSIGNED
ESP_BLE_MESH_ADDR_ALL_NODES
ESP_BLE_MESH_ADDR_PROXIES
ESP_BLE_MESH_ADDR_FRIENDS
ESP_BLE_MESH_ADDR_RELAYS
ESP_BLE_MESH_KEY_UNUSED
ESP_BLE_MESH_KEY_DEV
ESP_BLE_MESH_KEY_PRIMARY
ESP_BLE_MESH_KEY_ANY
Primary Network Key index
ESP_BLE_MESH_NET_PRIMARY
Relay state value
ESP_BLE_MESH_RELAY_DISABLED
ESP_BLE_MESH_RELAY_ENABLED
ESP_BLE_MESH_RELAY_NOT_SUPPORTED
Beacon state value
ESP_BLE_MESH_BEACON_DISABLED
ESP_BLE_MESH_BEACON_ENABLED
GATT Proxy state value
ESP_BLE_MESH_GATT_PROXY_DISABLED
ESP_BLE_MESH_GATT_PROXY_ENABLED
ESP_BLE_MESH_GATT_PROXY_NOT_SUPPORTED
Friend state value
ESP_BLE_MESH_FRIEND_DISABLED
ESP_BLE_MESH_FRIEND_ENABLED
ESP_BLE_MESH_FRIEND_NOT_SUPPORTED
Node identity state value
ESP_BLE_MESH_NODE_IDENTITY_STOPPED
ESP_BLE_MESH_NODE_IDENTITY_RUNNING
ESP_BLE_MESH_NODE_IDENTITY_NOT_SUPPORTED
Supported features
ESP_BLE_MESH_FEATURE_RELAY
ESP_BLE_MESH_FEATURE_PROXY
ESP_BLE_MESH_FEATURE_FRIEND
ESP_BLE_MESH_FEATURE_LOW_POWER
ESP_BLE_MESH_FEATURE_ALL_SUPPORTED
ESP_BLE_MESH_ADDR_IS_UNICAST (addr)
ESP_BLE_MESH_ADDR_IS_GROUP (addr)
ESP_BLE_MESH_ADDR_IS_VIRTUAL (addr)
ESP_BLE_MESH_ADDR_IS_RFU (addr)
ESP_BLE_MESH_INVALID_NODE_INDEX

ESP_BLE_MESH_TRANSMIT (count, int_ms)

Encode transmission count & interval steps.

Note For example, ESP_BLE_MESH_TRANSMIT(2, 20) means that the message will be sent about 90ms(count is 3, step is 1, interval is 30 ms which includes 10ms of advertising interval random delay).

Return BLE Mesh transmit value that can be used e.g. for the default values of the Configuration Model data.

Parameters

- `count`: Number of retransmissions (first transmission is excluded).
- `int_ms`: Interval steps in milliseconds. Must be greater than 0 and a multiple of 10.

ESP_BLE_MESH_GET_TRANSMIT_COUNT (transmit)

Decode transmit count from a transmit value.

Return Transmission count (actual transmissions equal to N + 1).

Parameters

- `transmit`: Encoded transmit count & interval value.

ESP_BLE_MESH_GET_TRANSMIT_INTERVAL (transmit)

Decode transmit interval from a transmit value.

Return Transmission interval in milliseconds.

Parameters

- `transmit`: Encoded transmit count & interval value.

ESP_BLE_MESH_PUBLISH_TRANSMIT (count, int_ms)

Encode Publish Retransmit count & interval steps.

Return BLE Mesh transmit value that can be used e.g. for the default values of the Configuration Model data.

Parameters

- `count`: Number of retransmissions (first transmission is excluded).
- `int_ms`: Interval steps in milliseconds. Must be greater than 0 and a multiple of 50.

ESP_BLE_MESH_GET_PUBLISH_TRANSMIT_COUNT (transmit)

Decode Publish Retransmit count from a given value.

Return Retransmission count (actual transmissions equal to N + 1).

Parameters

- `transmit`: Encoded Publish Retransmit count & interval value.

ESP_BLE_MESH_GET_PUBLISH_TRANSMIT_INTERVAL (transmit)

Decode Publish Retransmit interval from a given value.

Callbacks which are not needed to be initialized by users (set with 0 and will be initialized internally)

Return Transmission interval in milliseconds.

Parameters

- `transmit`: Encoded Publish Retransmit count & interval value.

ESP_BLE_MESH_PROV_STATIC_OOB_MAX_LEN

Maximum length of string used by Output OOB authentication

ESP_BLE_MESH_PROV_OUTPUT_OOB_MAX_LEN

Maximum length of string used by Output OOB authentication

ESP_BLE_MESH_PROV_INPUT_OOB_MAX_LEN

Macros used to define message opcode

ESP_BLE_MESH_MODEL_OP_1 (b0)

ESP_BLE_MESH_MODEL_OP_2 (b0, b1)

ESP_BLE_MESH_MODEL_OP_3 (b0, cid)

This macro is associated with BLE_MESH_MODEL_CB in mesh_access.h

ESP_BLE_MESH_SIG_MODEL (_id, _op, _pub, _user_data)

This macro is associated with BLE_MESH_MODEL_VND_CB in mesh_access.h

ESP_BLE_MESH_VENDOR_MODEL (_company, _id, _op, _pub, _user_data)

ESP_BLE_MESH_ELEMENT (_loc, _mods, _vnd_mods)

Helper to define a BLE Mesh element within an array.

In case the element has no SIG or Vendor models, the helper macro **ESP_BLE_MESH_MODEL_NONE** can be given instead.

Note This macro is associated with **BLE_MESH_ELEM** in **mesh_access.h**

Parameters

- **_loc**: Location Descriptor.
- **_mods**: Array of SIG models.
- **_vnd_mods**: Array of vendor models.

ESP_BLE_MESH_PROV (uuid, sta_val, sta_val_len, out_size, out_act, in_size, in_act)

BT_OCTET32_LEN

BD_ADDR_LEN

ESP_BLE_MESH_ADDR_TYPE_PUBLIC

ESP_BLE_MESH_ADDR_TYPE_RANDOM

ESP_BLE_MESH_ADDR_TYPE_RPA_PUBLIC

ESP_BLE_MESH_ADDR_TYPE_RPA_RANDOM

ESP_BLE_MESH_MODEL_PUB_DEFINE (_name, _msg_len, _role)

Define a model publication context.

Parameters

- **_name**: Variable name given to the context.
- **_msg_len**: Length of the publication message.
- **_role**: Role of the device which contains the model.

ESP_BLE_MESH_MODEL_OP (_opcode, _min_len)

Define a model operation context.

Parameters

- **_opcode**: Message opcode.
- **_min_len**: Message minimum length.

ESP_BLE_MESH_MODEL_OP_END

Define the terminator for the model operation table. Each model operation struct array must use this terminator as the end tag of the operation unit.

ESP_BLE_MESH_MODEL_NONE

Helper to define an empty model array. This structure is associated with **BLE_MESH_MODEL_NONE** in **mesh_access.h**

ADD_DEV_RM_AFTER_PROV_FLAG

Device will be removed from queue after provisioned successfully

ADD_DEV_START_PROV_NOW_FLAG

Start provisioning device immediately

ADD_DEV_FLUSHABLE_DEV_FLAG

Device can be remove when queue is full and new device is going to added

DEL_DEV_ADDR_FLAG

DEL_DEV_UUID_FLAG

PROV_DATA_NET_IDX_FLAG

PROV_DATA_FLAGS_FLAG

PROV_DATA_IV_INDEX_FLAG

ESP_BLE_MESH_HEARTBEAT_FILTER_ACCEPTLIST

ESP_BLE_MESH_HEARTBEAT_FILTER_REJECTLIST

Provisioner heartbeat filter operation

ESP_BLE_MESH_HEARTBEAT_FILTER_ADD

ESP_BLE_MESH_HEARTBEAT_FILTER_REMOVE

ESP_BLE_MESH_MODEL_ID_CONFIG_SRV

BLE Mesh models related Model ID and Opcode definitions.

< Foundation Models

ESP_BLE_MESH_MODEL_ID_CONFIG_CLI

ESP_BLE_MESH_MODEL_ID_HEALTH_SRV

ESP_BLE_MESH_MODEL_ID_HEALTH_CLI

Models from the Mesh Model Specification

ESP_BLE_MESH_MODEL_ID_GEN_ONOFF_SRV

ESP_BLE_MESH_MODEL_ID_GEN_ONOFF_CLI

ESP_BLE_MESH_MODEL_ID_GEN_LEVEL_SRV

ESP_BLE_MESH_MODEL_ID_GEN_LEVEL_CLI

ESP_BLE_MESH_MODEL_ID_GEN_DEF_TRANS_TIME_SRV

ESP_BLE_MESH_MODEL_ID_GEN_DEF_TRANS_TIME_CLI

ESP_BLE_MESH_MODEL_ID_GEN_POWER_ONOFF_SRV

ESP_BLE_MESH_MODEL_ID_GEN_POWER_ONOFF_SETUP_SRV

ESP_BLE_MESH_MODEL_ID_GEN_POWER_ONOFF_CLI

ESP_BLE_MESH_MODEL_ID_GEN_POWER_LEVEL_SRV

ESP_BLE_MESH_MODEL_ID_GEN_POWER_LEVEL_SETUP_SRV

ESP_BLE_MESH_MODEL_ID_GEN_POWER_LEVEL_CLI

ESP_BLE_MESH_MODEL_ID_GEN_BATTERY_SRV

ESP_BLE_MESH_MODEL_ID_GEN_BATTERY_CLI

ESP_BLE_MESH_MODEL_ID_GEN_LOCATION_SRV

ESP_BLE_MESH_MODEL_ID_GEN_LOCATION_SETUP_SRV

ESP_BLE_MESH_MODEL_ID_GEN_LOCATION_CLI

ESP_BLE_MESH_MODEL_ID_GEN_ADMIN_PROP_SRV

ESP_BLE_MESH_MODEL_ID_GEN_MANUFACTURER_PROP_SRV

ESP_BLE_MESH_MODEL_ID_GEN_USER_PROP_SRV

ESP_BLE_MESH_MODEL_ID_GEN_CLIENT_PROP_SRV

ESP_BLE_MESH_MODEL_ID_GEN_PROP_CLI

ESP_BLE_MESH_MODEL_ID_SENSOR_SRV

ESP_BLE_MESH_MODEL_ID_SENSOR_SETUP_SRV

ESP_BLE_MESH_MODEL_ID_SENSOR_CLI

ESP_BLE_MESH_MODEL_ID_TIME_SRV

ESP_BLE_MESH_MODEL_ID_TIME_SETUP_SRV

ESP_BLE_MESH_MODEL_ID_TIME_CLI

ESP_BLE_MESH_MODEL_ID_SCENE_SRV

ESP_BLE_MESH_MODEL_ID_SCENE_SETUP_SRV
ESP_BLE_MESH_MODEL_ID_SCENE_CLI
ESP_BLE_MESH_MODEL_ID_SCHEDULER_SRV
ESP_BLE_MESH_MODEL_ID_SCHEDULER_SETUP_SRV
ESP_BLE_MESH_MODEL_ID_SCHEDULER_CLI
ESP_BLE_MESH_MODEL_ID_LIGHT_LIGHTNESS_SRV
ESP_BLE_MESH_MODEL_ID_LIGHT_LIGHTNESS_SETUP_SRV
ESP_BLE_MESH_MODEL_ID_LIGHT_LIGHTNESS_CLI
ESP_BLE_MESH_MODEL_ID_LIGHT_CTL_SRV
ESP_BLE_MESH_MODEL_ID_LIGHT_CTL_SETUP_SRV
ESP_BLE_MESH_MODEL_ID_LIGHT_CTL_CLI
ESP_BLE_MESH_MODEL_ID_LIGHT_CTL_TEMP_SRV
ESP_BLE_MESH_MODEL_ID_LIGHT_HSL_SRV
ESP_BLE_MESH_MODEL_ID_LIGHT_HSL_SETUP_SRV
ESP_BLE_MESH_MODEL_ID_LIGHT_HSL_CLI
ESP_BLE_MESH_MODEL_ID_LIGHT_HSL_HUE_SRV
ESP_BLE_MESH_MODEL_ID_LIGHT_HSL_SAT_SRV
ESP_BLE_MESH_MODEL_ID_LIGHT_XYL_SRV
ESP_BLE_MESH_MODEL_ID_LIGHT_XYL_SETUP_SRV
ESP_BLE_MESH_MODEL_ID_LIGHT_XYL_CLI
ESP_BLE_MESH_MODEL_ID_LIGHT_LC_SRV
ESP_BLE_MESH_MODEL_ID_LIGHT_LC_SETUP_SRV
ESP_BLE_MESH_MODEL_ID_LIGHT_LC_CLI
ESP_BLE_MESH_MODEL_OP_BEACON_GET
Config Beacon Get
ESP_BLE_MESH_MODEL_OP_COMPOSITION_DATA_GET
Config Composition Data Get
ESP_BLE_MESH_MODEL_OP_DEFAULT_TTL_GET
Config Default TTL Get
ESP_BLE_MESH_MODEL_OP_GATT_PROXY_GET
Config GATT Proxy Get
ESP_BLE_MESH_MODEL_OP_RELAY_GET
Config Relay Get
ESP_BLE_MESH_MODEL_OP_MODEL_PUB_GET
Config Model Publication Get
ESP_BLE_MESH_MODEL_OP_FRIEND_GET
Config Friend Get
ESP_BLE_MESH_MODEL_OP_HEARTBEAT_PUB_GET
Config Heartbeat Publication Get
ESP_BLE_MESH_MODEL_OP_HEARTBEAT_SUB_GET
Config Heartbeat Subscription Get

ESP_BLE_MESH_MODEL_OP_NET_KEY_GET

Config NetKey Get

ESP_BLE_MESH_MODEL_OP_APP_KEY_GET

Config AppKey Get

ESP_BLE_MESH_MODEL_OP_NODE_IDENTITY_GET

Config Node Identity Get

ESP_BLE_MESH_MODEL_OP_SIG_MODEL_SUB_GET

Config SIG Model Subscription Get

ESP_BLE_MESH_MODEL_OP_VENDOR_MODEL_SUB_GET

Config Vendor Model Subscription Get

ESP_BLE_MESH_MODEL_OP_SIG_MODEL_APP_GET

Config SIG Model App Get

ESP_BLE_MESH_MODEL_OP_VENDOR_MODEL_APP_GET

Config Vendor Model App Get

ESP_BLE_MESH_MODEL_OP_KEY_REFRESH_PHASE_GET

Config Key Refresh Phase Get

ESP_BLE_MESH_MODEL_OP_LPN_POLLTIMEOUT_GET

Config Low Power Node PollTimeout Get

ESP_BLE_MESH_MODEL_OP_NETWORK_TRANSMIT_GET

Config Network Transmit Get

ESP_BLE_MESH_MODEL_OP_BEACON_SET

Config Beacon Set

ESP_BLE_MESH_MODEL_OP_DEFAULT_TTL_SET

Config Default TTL Set

ESP_BLE_MESH_MODEL_OP_GATT_PROXY_SET

Config GATT Proxy Set

ESP_BLE_MESH_MODEL_OP_RELAY_SET

Config Relay Set

ESP_BLE_MESH_MODEL_OP_MODEL_PUB_SET

Config Model Publication Set

ESP_BLE_MESH_MODEL_OP_MODEL_SUB_ADD

Config Model Subscription Add

ESP_BLE_MESH_MODEL_OP_MODEL_SUB_VIRTUAL_ADDR_ADD

Config Model Subscription Virtual Address Add

ESP_BLE_MESH_MODEL_OP_MODEL_SUB_DELETE

Config Model Subscription Delete

ESP_BLE_MESH_MODEL_OP_MODEL_SUB_VIRTUAL_ADDR_DELETE

Config Model Subscription Virtual Address Delete

ESP_BLE_MESH_MODEL_OP_MODEL_SUB_OVERWRITE

Config Model Subscription Overwrite

ESP_BLE_MESH_MODEL_OP_MODEL_SUB_VIRTUAL_ADDR_OVERWRITE

Config Model Subscription Virtual Address Overwrite

ESP_BLE_MESH_MODEL_OP_NET_KEY_ADD

Config NetKey Add

ESP_BLE_MESH_MODEL_OP_APP_KEY_ADD

Config AppKey Add

ESP_BLE_MESH_MODEL_OP_MODEL_APP_BIND
Config Model App Bind

ESP_BLE_MESH_MODEL_OP_NODE_RESET
Config Node Reset

ESP_BLE_MESH_MODEL_OP_FRIEND_SET
Config Friend Set

ESP_BLE_MESH_MODEL_OP_HEARTBEAT_PUB_SET
Config Heartbeat Publication Set

ESP_BLE_MESH_MODEL_OP_HEARTBEAT_SUB_SET
Config Heartbeat Subscription Set

ESP_BLE_MESH_MODEL_OP_NET_KEY_UPDATE
Config NetKey Update

ESP_BLE_MESH_MODEL_OP_NET_KEY_DELETE
Config NetKey Delete

ESP_BLE_MESH_MODEL_OP_APP_KEY_UPDATE
Config AppKey Update

ESP_BLE_MESH_MODEL_OP_APP_KEY_DELETE
Config AppKey Delete

ESP_BLE_MESH_MODEL_OP_NODE_IDENTITY_SET
Config Node Identity Set

ESP_BLE_MESH_MODEL_OP_KEY_REFRESH_PHASE_SET
Config Key Refresh Phase Set

ESP_BLE_MESH_MODEL_OP_MODEL_PUB_VIRTUAL_ADDR_SET
Config Model Publication Virtual Address Set

ESP_BLE_MESH_MODEL_OP_MODEL_SUB_DELETE_ALL
Config Model Subscription Delete All

ESP_BLE_MESH_MODEL_OP_MODEL_APP_UNBIND
Config Model App Unbind

ESP_BLE_MESH_MODEL_OP_NETWORK_TRANSMIT_SET
Config Network Transmit Set

ESP_BLE_MESH_MODEL_OP_BEACON_STATUS

ESP_BLE_MESH_MODEL_OP_COMPOSITION_DATA_STATUS

ESP_BLE_MESH_MODEL_OP_DEFAULT_TTL_STATUS

ESP_BLE_MESH_MODEL_OP_GATT_PROXY_STATUS

ESP_BLE_MESH_MODEL_OP_RELAY_STATUS

ESP_BLE_MESH_MODEL_OP_MODEL_PUB_STATUS

ESP_BLE_MESH_MODEL_OP_MODEL_SUB_STATUS

ESP_BLE_MESH_MODEL_OP_SIG_MODEL_SUB_LIST

ESP_BLE_MESH_MODEL_OP_VENDOR_MODEL_SUB_LIST

ESP_BLE_MESH_MODEL_OP_NET_KEY_STATUS

ESP_BLE_MESH_MODEL_OP_NET_KEY_LIST

ESP_BLE_MESH_MODEL_OP_APP_KEY_STATUS

ESP_BLE_MESH_MODEL_OP_APP_KEY_LIST

ESP_BLE_MESH_MODEL_OP_NODE_IDENTITY_STATUS

ESP_BLE_MESH_MODEL_OP_MODEL_APP_STATUS
ESP_BLE_MESH_MODEL_OP_SIG_MODEL_APP_LIST
ESP_BLE_MESH_MODEL_OP_VENDOR_MODEL_APP_LIST
ESP_BLE_MESH_MODEL_OP_NODE_RESET_STATUS
ESP_BLE_MESH_MODEL_OP_FRIEND_STATUS
ESP_BLE_MESH_MODEL_OP_KEY_REFRESH_PHASE_STATUS
ESP_BLE_MESH_MODEL_OP_HEARTBEAT_PUB_STATUS
ESP_BLE_MESH_MODEL_OP_HEARTBEAT_SUB_STATUS
ESP_BLE_MESH_MODEL_OP_LPN_POLLTIMEOUT_STATUS
ESP_BLE_MESH_MODEL_OP_NETWORK_TRANSMIT_STATUS
ESP_BLE_MESH_CFG_STATUS_SUCCESS
ESP_BLE_MESH_CFG_STATUS_INVALID_ADDRESS
ESP_BLE_MESH_CFG_STATUS_INVALID_MODEL
ESP_BLE_MESH_CFG_STATUS_INVALID_APPKEY
ESP_BLE_MESH_CFG_STATUS_INVALID_NETKEY
ESP_BLE_MESH_CFG_STATUS_INSUFFICIENT_RESOURCES
ESP_BLE_MESH_CFG_STATUS_KEY_INDEX_ALREADY_STORED
ESP_BLE_MESH_CFG_STATUS_INVALID_PUBLISH_PARAMETERS
ESP_BLE_MESH_CFG_STATUS_NOT_A_SUBSCRIBE_MODEL
ESP_BLE_MESH_CFG_STATUS_STORAGE_FAILURE
ESP_BLE_MESH_CFG_STATUS_FEATURE_NOT_SUPPORTED
ESP_BLE_MESH_CFG_STATUS_CANNOT_UPDATE
ESP_BLE_MESH_CFG_STATUS_CANNOT_REMOVE
ESP_BLE_MESH_CFG_STATUS_CANNOT_BIND
ESP_BLE_MESH_CFG_STATUS_TEMP_UNABLE_TO_CHANGE_STATE
ESP_BLE_MESH_CFG_STATUS_CANNOT_SET
ESP_BLE_MESH_CFG_STATUS_UNSPECIFIED_ERROR
ESP_BLE_MESH_CFG_STATUS_INVALID_BINDING
ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_GET
Health Fault Get
ESP_BLE_MESH_MODEL_OP_HEALTH_PERIOD_GET
Health Period Get
ESP_BLE_MESH_MODEL_OP_ATTENTION_GET
Health Attention Get
ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_CLEAR
Health Fault Clear
ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_CLEAR_UNACK
Health Fault Clear Unacknowledged
ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_TEST
Health Fault Test

ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_TEST_UNACK
Health Fault Test Unacknowledged

ESP_BLE_MESH_MODEL_OP_HEALTH_PERIOD_SET
Health Period Set

ESP_BLE_MESH_MODEL_OP_HEALTH_PERIOD_SET_UNACK
Health Period Set Unacknowledged

ESP_BLE_MESH_MODEL_OP_ATTENTION_SET
Health Attention Set

ESP_BLE_MESH_MODEL_OP_ATTENTION_SET_UNACK
Health Attention Set Unacknowledged

ESP_BLE_MESH_MODEL_OP_HEALTH_CURRENT_STATUS

ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_STATUS

ESP_BLE_MESH_MODEL_OP_HEALTH_PERIOD_STATUS

ESP_BLE_MESH_MODEL_OP_ATTENTION_STATUS

ESP_BLE_MESH_MODEL_OP_GEN_ONOFF_GET

ESP_BLE_MESH_MODEL_OP_GEN_ONOFF_SET

ESP_BLE_MESH_MODEL_OP_GEN_ONOFF_SET_UNACK

ESP_BLE_MESH_MODEL_OP_GEN_ONOFF_STATUS
Generic Level Message Opcode

ESP_BLE_MESH_MODEL_OP_GEN_LEVEL_GET

ESP_BLE_MESH_MODEL_OP_GEN_LEVEL_SET

ESP_BLE_MESH_MODEL_OP_GEN_LEVEL_SET_UNACK

ESP_BLE_MESH_MODEL_OP_GEN_LEVEL_STATUS

ESP_BLE_MESH_MODEL_OP_GEN_DELTA_SET

ESP_BLE_MESH_MODEL_OP_GEN_DELTA_SET_UNACK

ESP_BLE_MESH_MODEL_OP_GEN_MOVE_SET

ESP_BLE_MESH_MODEL_OP_GEN_MOVE_SET_UNACK
Generic Default Transition Time Message Opcode

ESP_BLE_MESH_MODEL_OP_GEN_DEF_TRANS_TIME_GET

ESP_BLE_MESH_MODEL_OP_GEN_DEF_TRANS_TIME_SET

ESP_BLE_MESH_MODEL_OP_GEN_DEF_TRANS_TIME_SET_UNACK

ESP_BLE_MESH_MODEL_OP_GEN_DEF_TRANS_TIME_STATUS
Generic Power OnOff Message Opcode

ESP_BLE_MESH_MODEL_OP_GEN_ONPOWERUP_GET

ESP_BLE_MESH_MODEL_OP_GEN_ONPOWERUP_STATUS
Generic Power OnOff Setup Message Opcode

ESP_BLE_MESH_MODEL_OP_GEN_ONPOWERUP_SET

ESP_BLE_MESH_MODEL_OP_GEN_ONPOWERUP_SET_UNACK
Generic Power Level Message Opcode

ESP_BLE_MESH_MODEL_OP_GEN_POWER_LEVEL_GET

ESP_BLE_MESH_MODEL_OP_GEN_POWER_LEVEL_SET

ESP_BLE_MESH_MODEL_OP_GEN_POWER_LEVEL_SET_UNACK

ESP_BLE_MESH_MODEL_OP_GEN_POWER_LEVEL_STATUS
ESP_BLE_MESH_MODEL_OP_GEN_POWER_LAST_GET
ESP_BLE_MESH_MODEL_OP_GEN_POWER_LAST_STATUS
ESP_BLE_MESH_MODEL_OP_GEN_POWER_DEFAULT_GET
ESP_BLE_MESH_MODEL_OP_GEN_POWER_DEFAULT_STATUS
ESP_BLE_MESH_MODEL_OP_GEN_POWER_RANGE_GET
ESP_BLE_MESH_MODEL_OP_GEN_POWER_RANGE_STATUS
Generic Power Level Setup Message Opcode
ESP_BLE_MESH_MODEL_OP_GEN_POWER_DEFAULT_SET
ESP_BLE_MESH_MODEL_OP_GEN_POWER_DEFAULT_SET_UNACK
ESP_BLE_MESH_MODEL_OP_GEN_POWER_RANGE_SET
ESP_BLE_MESH_MODEL_OP_GEN_POWER_RANGE_SET_UNACK
Generic Battery Message Opcode
ESP_BLE_MESH_MODEL_OP_GEN_BATTERY_GET
ESP_BLE_MESH_MODEL_OP_GEN_BATTERY_STATUS
Generic Location Message Opcode
ESP_BLE_MESH_MODEL_OP_GEN_LOC_GLOBAL_GET
ESP_BLE_MESH_MODEL_OP_GEN_LOC_GLOBAL_STATUS
ESP_BLE_MESH_MODEL_OP_GEN_LOC_LOCAL_GET
ESP_BLE_MESH_MODEL_OP_GEN_LOC_LOCAL_STATUS
Generic Location Setup Message Opcode
ESP_BLE_MESH_MODEL_OP_GEN_LOC_GLOBAL_SET
ESP_BLE_MESH_MODEL_OP_GEN_LOC_GLOBAL_SET_UNACK
ESP_BLE_MESH_MODEL_OP_GEN_LOC_LOCAL_SET
ESP_BLE_MESH_MODEL_OP_GEN_LOC_LOCAL_SET_UNACK
Generic Manufacturer Property Message Opcode
ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTIES_GET
ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTIES_STATUS
ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTY_GET
ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTY_SET
ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTY_SET_UNACK
ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTY_STATUS
Generic Admin Property Message Opcode
ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTIES_GET
ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTIES_STATUS
ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTY_GET
ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTY_SET
ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTY_SET_UNACK
ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTY_STATUS
Generic User Property Message Opcode
ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTIES_GET

ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTIES_STATUS
ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTY_GET
ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTY_SET
ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTY_SET_UNACK
ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTY_STATUS
Generic Client Property Message Opcode
ESP_BLE_MESH_MODEL_OP_GEN_CLIENT_PROPERTIES_GET
ESP_BLE_MESH_MODEL_OP_GEN_CLIENT_PROPERTIES_STATUS
ESP_BLE_MESH_MODEL_OP_SENSOR_DESCRIPTOR_GET
ESP_BLE_MESH_MODEL_OP_SENSOR_DESCRIPTOR_STATUS
ESP_BLE_MESH_MODEL_OP_SENSOR_GET
ESP_BLE_MESH_MODEL_OP_SENSOR_STATUS
ESP_BLE_MESH_MODEL_OP_SENSOR_COLUMN_GET
ESP_BLE_MESH_MODEL_OP_SENSOR_COLUMN_STATUS
ESP_BLE_MESH_MODEL_OP_SENSOR_SERIES_GET
ESP_BLE_MESH_MODEL_OP_SENSOR_SERIES_STATUS
Sensor Setup Message Opcode
ESP_BLE_MESH_MODEL_OP_SENSOR_CADENCE_GET
ESP_BLE_MESH_MODEL_OP_SENSOR_CADENCE_SET
ESP_BLE_MESH_MODEL_OP_SENSOR_CADENCE_SET_UNACK
ESP_BLE_MESH_MODEL_OP_SENSOR_CADENCE_STATUS
ESP_BLE_MESH_MODEL_OP_SENSOR_SETTINGS_GET
ESP_BLE_MESH_MODEL_OP_SENSOR_SETTINGS_STATUS
ESP_BLE_MESH_MODEL_OP_SENSOR_SETTING_GET
ESP_BLE_MESH_MODEL_OP_SENSOR_SETTING_SET
ESP_BLE_MESH_MODEL_OP_SENSOR_SETTING_SET_UNACK
ESP_BLE_MESH_MODEL_OP_SENSOR_SETTING_STATUS
ESP_BLE_MESH_MODEL_OP_TIME_GET
ESP_BLE_MESH_MODEL_OP_TIME_SET
ESP_BLE_MESH_MODEL_OP_TIME_STATUS
ESP_BLE_MESH_MODEL_OP_TIME_ROLE_GET
ESP_BLE_MESH_MODEL_OP_TIME_ROLE_SET
ESP_BLE_MESH_MODEL_OP_TIME_ROLE_STATUS
ESP_BLE_MESH_MODEL_OP_TIME_ZONE_GET
ESP_BLE_MESH_MODEL_OP_TIME_ZONE_SET
ESP_BLE_MESH_MODEL_OP_TIME_ZONE_STATUS
ESP_BLE_MESH_MODEL_OP_TAI.UTC_DELTA_GET
ESP_BLE_MESH_MODEL_OP_TAI.UTC_DELTA_SET
ESP_BLE_MESH_MODEL_OP_TAI.UTC_DELTA_STATUS
Scene Message Opcode

ESP_BLE_MESH_MODEL_OP_SCENE_GET
ESP_BLE_MESH_MODEL_OP_SCENE_RECALL
ESP_BLE_MESH_MODEL_OP_SCENE_RECALL_UNACK
ESP_BLE_MESH_MODEL_OP_SCENE_STATUS
ESP_BLE_MESH_MODEL_OP_SCENE_REGISTER_GET
ESP_BLE_MESH_MODEL_OP_SCENE_REGISTER_STATUS
Scene Setup Message Opcode
ESP_BLE_MESH_MODEL_OP_SCENE_STORE
ESP_BLE_MESH_MODEL_OP_SCENE_STORE_UNACK
ESP_BLE_MESH_MODEL_OP_SCENE_DELETE
ESP_BLE_MESH_MODEL_OP_SCENE_DELETE_UNACK
Scheduler Message Opcode
ESP_BLE_MESH_MODEL_OP_SCHEDULER_ACT_GET
ESP_BLE_MESH_MODEL_OP_SCHEDULER_ACT_STATUS
ESP_BLE_MESH_MODEL_OP_SCHEDULER_GET
ESP_BLE_MESH_MODEL_OP_SCHEDULER_STATUS
Scheduler Setup Message Opcode
ESP_BLE_MESH_MODEL_OP_SCHEDULER_ACT_SET
ESP_BLE_MESH_MODEL_OP_SCHEDULER_ACT_SET_UNACK
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_GET
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_SET
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_SET_UNACK
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_STATUS
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_LINEAR_GET
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_LINEAR_SET
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_LINEAR_SET_UNACK
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_LINEAR_STATUS
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_LAST_GET
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_LAST_STATUS
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_DEFAULT_GET
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_DEFAULT_STATUS
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_RANGE_GET
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_RANGE_STATUS
Light Lightness Setup Message Opcode
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_DEFAULT_SET
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_DEFAULT_SET_UNACK
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_RANGE_SET
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_RANGE_SET_UNACK
Light CTL Message Opcode
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_GET
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_SET

ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_SET_UNACK
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_STATUS
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_GET
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_RANGE_GET
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_RANGE_STATUS
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_SET
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_SET_UNACK
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_STATUS
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_DEFAULT_GET
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_DEFAULT_STATUS
Light CTL Setup Message Opcode
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_DEFAULT_SET
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_DEFAULT_SET_UNACK
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_RANGE_SET
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_RANGE_SET_UNACK
Light HSL Message Opcode
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_GET
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_HUE_GET
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_HUE_SET
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_HUE_SET_UNACK
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_HUE_STATUS
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_SATURATION_GET
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_SATURATION_SET
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_SATURATION_SET_UNACK
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_SATURATION_STATUS
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_SET
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_SET_UNACK
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_STATUS
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_TARGET_GET
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_TARGET_STATUS
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_DEFAULT_GET
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_DEFAULT_STATUS
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_RANGE_GET
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_RANGE_STATUS
Light HSL Setup Message Opcode
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_DEFAULT_SET
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_DEFAULT_SET_UNACK
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_RANGE_SET
ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_RANGE_SET_UNACK
Light xyL Message Opcode

ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_GET
ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_SET
ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_SET_UNACK
ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_STATUS
ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_TARGET_GET
ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_TARGET_STATUS
ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_DEFAULT_GET
ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_DEFAULT_STATUS
ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_RANGE_GET
ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_RANGE_STATUS
 Light xyL Setup Message Opcode
ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_DEFAULT_SET
ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_DEFAULT_SET_UNACK
ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_RANGE_SET
ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_RANGE_SET_UNACK
 Light Control Message Opcode
ESP_BLE_MESH_MODEL_OP_LIGHT_LC_MODE_GET
ESP_BLE_MESH_MODEL_OP_LIGHT_LC_MODE_SET
ESP_BLE_MESH_MODEL_OP_LIGHT_LC_MODE_SET_UNACK
ESP_BLE_MESH_MODEL_OP_LIGHT_LC_MODE_STATUS
ESP_BLE_MESH_MODEL_OP_LIGHT_LC_OM_GET
ESP_BLE_MESH_MODEL_OP_LIGHT_LC_OM_SET
ESP_BLE_MESH_MODEL_OP_LIGHT_LC_OM_SET_UNACK
ESP_BLE_MESH_MODEL_OP_LIGHT_LC_OM_STATUS
ESP_BLE_MESH_MODEL_OP_LIGHT_LC_LIGHT_ONOFF_GET
ESP_BLE_MESH_MODEL_OP_LIGHT_LC_LIGHT_ONOFF_SET
ESP_BLE_MESH_MODEL_OP_LIGHT_LC_LIGHT_ONOFF_SET_UNACK
ESP_BLE_MESH_MODEL_OP_LIGHT_LC_LIGHT_ONOFF_STATUS
ESP_BLE_MESH_MODEL_OP_LIGHT_LC_PROPERTY_GET
ESP_BLE_MESH_MODEL_OP_LIGHT_LC_PROPERTY_SET
ESP_BLE_MESH_MODEL_OP_LIGHT_LC_PROPERTY_SET_UNACK
ESP_BLE_MESH_MODEL_OP_LIGHT_LC_PROPERTY_STATUS
ESP_BLE_MESH_MODEL_STATUS_SUCCESS
ESP_BLE_MESH_MODEL_STATUS_CANNOT_SET_RANGE_MIN
ESP_BLE_MESH_MODEL_STATUS_CANNOT_SET_RANGE_MAX
ESP_BLE_MESH_SERVER_RSP_BY_APP
 Response need to be sent in the application
ESP_BLE_MESH_SERVER_AUTO_RSP
 Response will be sent internally

Type Definitions

typedef uint8_t **esp_ble_mesh_octet16_t**[ESP_BLE_MESH_OCTET16_LEN]

Define the BLE Mesh octet 8 bytes size

typedef uint8_t **esp_ble_mesh_octet8_t**[ESP_BLE_MESH_OCTET8_LEN]

Invalid Company ID

typedef uint32_t **esp_ble_mesh_cb_t**

typedef uint8_t **UINT8**

typedef uint16_t **UINT16**

typedef uint32_t **UINT32**

typedef uint64_t **UINT64**

typedef *UINT8* **BT_OCTET32**[BT_OCTET32_LEN]

typedef uint8_t **BD_ADDR**[BD_ADDR_LEN]

typedef uint8_t **esp_ble_mesh_bd_addr_t**[BD_ADDR_LEN]

typedef uint8_t **esp_ble_mesh_addr_type_t**

BLE device address type.

typedef struct *esp_ble_mesh_model* **esp_ble_mesh_model_t**

typedef uint8_t **esp_ble_mesh_dev_add_flag_t**

typedef uint32_t **esp_ble_mesh_opcode_config_client_get_t**

esp_ble_mesh_opcode_config_client_get_t belongs to **esp_ble_mesh_opcode_t**, this typedef is only used to locate the opcodes used by **esp_ble_mesh_config_client_get_state**. The following opcodes will only be used in the **esp_ble_mesh_config_client_get_state** function.

typedef uint32_t **esp_ble_mesh_opcode_config_client_set_t**

esp_ble_mesh_opcode_config_client_set_t belongs to **esp_ble_mesh_opcode_t**, this typedef is only used to locate the opcodes used by **esp_ble_mesh_config_client_set_state**. The following opcodes will only be used in the **esp_ble_mesh_config_client_set_state** function.

typedef uint32_t **esp_ble_mesh_opcode_config_status_t**

esp_ble_mesh_opcode_config_status_t belongs to **esp_ble_mesh_opcode_t**, this typedef is only used to locate the opcodes used by the Config Model messages. The following opcodes are used by the BLE Mesh Config Server Model internally to respond to the Config Client Model's request messages.

typedef uint8_t **esp_ble_mesh_cfg_status_t**

This typedef is only used to indicate the status code contained in some of the Configuration Server Model status message.

typedef uint32_t **esp_ble_mesh_opcode_health_client_get_t**

esp_ble_mesh_opcode_health_client_get_t belongs to **esp_ble_mesh_opcode_t**, this typedef is only used to locate the opcodes used by **esp_ble_mesh_health_client_get_state**. The following opcodes will only be used in the **esp_ble_mesh_health_client_get_state** function.

typedef uint32_t **esp_ble_mesh_opcode_health_client_set_t**

esp_ble_mesh_opcode_health_client_set_t belongs to **esp_ble_mesh_opcode_t**, this typedef is only used to locate the opcodes used by **esp_ble_mesh_health_client_set_state**. The following opcodes will only be used in the **esp_ble_mesh_health_client_set_state** function.

typedef uint32_t **esp_ble_mesh_health_model_status_t**

esp_ble_mesh_health_model_status_t belongs to **esp_ble_mesh_opcode_t**, this typedef is only used to locate the opcodes used by the Health Model messages. The following opcodes are used by the BLE Mesh Health Server Model internally to respond to the Health Client Model's request messages.

typedef uint32_t **esp_ble_mesh_generic_message_opcode_t**

esp_ble_mesh_generic_message_opcode_t belongs to **esp_ble_mesh_opcode_t**, this typedef is only used to locate the opcodes used by functions **esp_ble_mesh_generic_client_get_state** & **esp_ble_mesh_generic_client_set_state**. Generic OnOff Message Opcode

typedef uint32_t **esp_ble_mesh_sensor_message_opcode_t**

esp_ble_mesh_sensor_message_opcode_t belongs to esp_ble_mesh_opcode_t, this typedef is only used to locate the opcodes used by functions esp_ble_mesh_sensor_client_get_state & esp_ble_mesh_sensor_client_set_state.Sensor Message Opcode

typedef uint32_t **esp_ble_mesh_time_scene_message_opcode_t**

esp_ble_mesh_time_scene_message_opcode_t belongs to esp_ble_mesh_opcode_t, this typedef is only used to locate the opcodes used by functions esp_ble_mesh_time_scene_client_get_state & esp_ble_mesh_time_scene_client_set_state.Time Message Opcode

typedef uint32_t **esp_ble_mesh_light_message_opcode_t**

esp_ble_mesh_light_message_opcode_t belongs to esp_ble_mesh_opcode_t, this typedef is only used to locate the opcodes used by functions esp_ble_mesh_light_client_get_state & esp_ble_mesh_light_client_set_state.Light Lightness Message Opcode

typedef uint32_t **esp_ble_mesh_opcode_t**

End of defines of esp_ble_mesh_opcode_t

typedef uint8_t **esp_ble_mesh_model_status_t**

This typedef is only used to indicate the status code contained in some of the server models (e.g. Generic Server Model) status message.

Enumerations

enum esp_ble_mesh_cb_type_t

Values:

ESP_BLE_MESH_TYPE_PROV_CB

ESP_BLE_MESH_TYPE_OUTPUT_NUM_CB

ESP_BLE_MESH_TYPE_OUTPUT_STR_CB

ESP_BLE_MESH_TYPE_INTPUT_CB

ESP_BLE_MESH_TYPE_LINK_OPEN_CB

ESP_BLE_MESH_TYPE_LINK_CLOSE_CB

ESP_BLE_MESH_TYPE_COMPLETE_CB

ESP_BLE_MESH_TYPE_RESET_CB

enum esp_ble_mesh_oob_method_t

Values:

ESP_BLE_MESH_NO_OOB

ESP_BLE_MESH_STATIC_OOB

ESP_BLE_MESH_OUTPUT_OOB

ESP_BLE_MESH_INPUT_OOB

enum esp_ble_mesh_output_action_t

Values:

ESP_BLE_MESH_NO_OUTPUT = 0

ESP_BLE_MESH_BLINK = BIT(0)

ESP_BLE_MESH_BEEP = BIT(1)

ESP_BLE_MESH_VIBRATE = BIT(2)

ESP_BLE_MESH_DISPLAY_NUMBER = BIT(3)

ESP_BLE_MESH_DISPLAY_STRING = BIT(4)

enum esp_ble_mesh_input_action_t

Values:

ESP_BLE_MESH_NO_INPUT = 0

ESP_BLE_MESH_PUSH = BIT(0)

ESP_BLE_MESH_TWIST = BIT(1)

ESP_BLE_MESH_ENTER_NUMBER = BIT(2)

ESP_BLE_MESH_ENTER_STRING = BIT(3)

enum esp_ble_mesh_prov_bearer_t

Values:

ESP_BLE_MESH_PROV_ADV = BIT(0)

ESP_BLE_MESH_PROV_GATT = BIT(1)

enum esp_ble_mesh_prov_oob_info_t

Values:

ESP_BLE_MESH_PROV_OOB_OTHER = BIT(0)

ESP_BLE_MESH_PROV_OOB_URI = BIT(1)

ESP_BLE_MESH_PROV_OOB_2D_CODE = BIT(2)

ESP_BLE_MESH_PROV_OOB_BAR_CODE = BIT(3)

ESP_BLE_MESH_PROV_OOB_NFC = BIT(4)

ESP_BLE_MESH_PROV_OOB_NUMBER = BIT(5)

ESP_BLE_MESH_PROV_OOB_STRING = BIT(6)

ESP_BLE_MESH_PROV_OOB_ON_BOX = BIT(11)

ESP_BLE_MESH_PROV_OOB_IN_BOX = BIT(12)

ESP_BLE_MESH_PROV_OOB_ON_PAPER = BIT(13)

ESP_BLE_MESH_PROV_OOB_IN_MANUAL = BIT(14)

ESP_BLE_MESH_PROV_OOB_ON_DEV = BIT(15)

enum esp_ble_mesh_dev_role_t

Values:

ROLE_NODE = 0

ROLE_PROVISIONER

ROLE_FAST_PROV

enum esp_ble_mesh_fast_prov_action_t

Values:

FAST_PROV_ACT_NONE

FAST_PROV_ACT_ENTER

FAST_PROV_ACT_SUSPEND

FAST_PROV_ACT_EXIT

FAST_PROV_ACT_MAX

enum esp_ble_mesh_proxy_filter_type_t

Values:

PROXY_FILTER_WHITELIST

PROXY_FILTER_BLACKLIST

enum esp_ble_mesh_prov_cb_event_t

Values:

| | |
|---|--|
| ESP_BLE_MESH_PROV_REGISTER_COMP_EVT | Initialize BLE Mesh provisioning capabilities and internal data information completion event |
| ESP_BLE_MESH_NODE_SET_UNPROV_DEV_NAME_COMP_EVT | Set the unprovisioned device name completion event |
| ESP_BLE_MESH_NODE_PROV_ENABLE_COMP_EVT | Enable node provisioning functionality completion event |
| ESP_BLE_MESH_NODE_PROV_DISABLE_COMP_EVT | Disable node provisioning functionality completion event |
| ESP_BLE_MESH_NODE_PROV_LINK_OPEN_EVT | Establish a BLE Mesh link event |
| ESP_BLE_MESH_NODE_PROV_LINK_CLOSE_EVT | Close a BLE Mesh link event |
| ESP_BLE_MESH_NODE_PROV_OOB_PUB_KEY_EVT | Generate Node input OOB public key event |
| ESP_BLE_MESH_NODE_PROV_OUTPUT_NUMBER_EVT | Generate Node Output Number event |
| ESP_BLE_MESH_NODE_PROV_OUTPUT_STRING_EVT | Generate Node Output String event |
| ESP_BLE_MESH_NODE_PROV_INPUT_EVT | Event requiring the user to input a number or string |
| ESP_BLE_MESH_NODE_PROV_COMPLETE_EVT | Provisioning done event |
| ESP_BLE_MESH_NODE_PROV_RESET_EVT | Provisioning reset event |
| ESP_BLE_MESH_NODE_PROV_SET_OOB_PUB_KEY_COMP_EVT | Node set oob public key completion event |
| ESP_BLE_MESH_NODE_PROV_INPUT_NUMBER_COMP_EVT | Node input number completion event |
| ESP_BLE_MESH_NODE_PROV_INPUT_STRING_COMP_EVT | Node input string completion event |
| ESP_BLE_MESH_NODE_PROXY_IDENTITY_ENABLE_COMP_EVT | Enable BLE Mesh Proxy Identity advertising completion event |
| ESP_BLE_MESH_NODE_PROXY_GATT_ENABLE_COMP_EVT | Enable BLE Mesh GATT Proxy Service completion event |
| ESP_BLE_MESH_NODE_PROXY_GATT_DISABLE_COMP_EVT | Disable BLE Mesh GATT Proxy Service completion event |
| ESP_BLE_MESH_NODE_ADD_LOCAL_NET_KEY_COMP_EVT | Node add NetKey locally completion event |
| ESP_BLE_MESH_NODE_ADD_LOCAL_APP_KEY_COMP_EVT | Node add AppKey locally completion event |
| ESP_BLE_MESH_NODE_BIND_APP_KEY_TO_MODEL_COMP_EVT | Node bind AppKey to model locally completion event |
| ESP_BLE_MESH_PROVISIONER_PROV_ENABLE_COMP_EVT | Provisioner enable provisioning functionality completion event |
| ESP_BLE_MESH_PROVISIONER_PROV_DISABLE_COMP_EVT | Provisioner disable provisioning functionality completion event |

- ESP_BLE_MESH_PROVISIONER_RECV_UNPROV_ADV_PKT_EVT**
Provisioner receives unprovisioned device beacon event
- ESP_BLE_MESH_PROVISIONER_PROV_READ_OOB_PUB_KEY_EVT**
Provisioner read unprovisioned device OOB public key event
- ESP_BLE_MESH_PROVISIONER_PROV_INPUT_EVT**
Provisioner input value for provisioning procedure event
- ESP_BLE_MESH_PROVISIONER_PROV_OUTPUT_EVT**
Provisioner output value for provisioning procedure event
- ESP_BLE_MESH_PROVISIONER_PROV_LINK_OPEN_EVT**
Provisioner establish a BLE Mesh link event
- ESP_BLE_MESH_PROVISIONER_PROV_LINK_CLOSE_EVT**
Provisioner close a BLE Mesh link event
- ESP_BLE_MESH_PROVISIONER_PROV_COMPLETE_EVT**
Provisioner provisioning done event
- ESP_BLE_MESH_PROVISIONER_ADD_UNPROV_DEV_COMP_EVT**
Provisioner add a device to the list which contains devices that are waiting/going to be provisioned completion event
- ESP_BLE_MESH_PROVISIONER_PROV_DEV_WITH_ADDR_COMP_EVT**
Provisioner start to provision an unprovisioned device completion event
- ESP_BLE_MESH_PROVISIONER_DELETE_DEV_COMP_EVT**
Provisioner delete a device from the list, close provisioning link with the device completion event
- ESP_BLE_MESH_PROVISIONER_SET_DEV_UUID_MATCH_COMP_EVT**
Provisioner set the value to be compared with part of the unprovisioned device UUID completion event
- ESP_BLE_MESH_PROVISIONER_SET_PROV_DATA_INFO_COMP_EVT**
Provisioner set net_idx/flags/iv_index used for provisioning completion event
- ESP_BLE_MESH_PROVISIONER_SET_STATIC_OOB_VALUE_COMP_EVT**
Provisioner set static oob value used for provisioning completion event
- ESP_BLE_MESH_PROVISIONER_SET_PRIMARY_ELEM_ADDR_COMP_EVT**
Provisioner set unicast address of primary element completion event
- ESP_BLE_MESH_PROVISIONER_PROV_READ_OOB_PUB_KEY_COMP_EVT**
Provisioner read unprovisioned device OOB public key completion event
- ESP_BLE_MESH_PROVISIONER_PROV_INPUT_NUMBER_COMP_EVT**
Provisioner input number completion event
- ESP_BLE_MESH_PROVISIONER_PROV_INPUT_STRING_COMP_EVT**
Provisioner input string completion event
- ESP_BLE_MESH_PROVISIONER_SET_NODE_NAME_COMP_EVT**
Provisioner set node name completion event
- ESP_BLE_MESH_PROVISIONER_ADD_LOCAL_APP_KEY_COMP_EVT**
Provisioner add local app key completion event
- ESP_BLE_MESH_PROVISIONER_UPDATE_LOCAL_APP_KEY_COMP_EVT**
Provisioner update local app key completion event
- ESP_BLE_MESH_PROVISIONER_BIND_APP_KEY_TO_MODEL_COMP_EVT**
Provisioner bind local model with local app key completion event
- ESP_BLE_MESH_PROVISIONER_ADD_LOCAL_NET_KEY_COMP_EVT**
Provisioner add local network key completion event
- ESP_BLE_MESH_PROVISIONER_UPDATE_LOCAL_NET_KEY_COMP_EVT**
Provisioner update local network key completion event

ESP_BLE_MESH_PROVISIONER_STORE_NODE_COMP_DATA_COMP_EVT
Provisioner store node composition data completion event

ESP_BLE_MESH_PROVISIONER_DELETE_NODE_WITH_UUID_COMP_EVT
Provisioner delete node with uuid completion event

ESP_BLE_MESH_PROVISIONER_DELETE_NODE_WITH_ADDR_COMP_EVT
Provisioner delete node with unicast address completion event

ESP_BLE_MESH_PROVISIONER_ENABLE_HEARTBEAT_RECV_COMP_EVT
Provisioner start to receive heartbeat message completion event

ESP_BLE_MESH_PROVISIONER_SET_HEARTBEAT_FILTER_TYPE_COMP_EVT
Provisioner set the heartbeat filter type completion event

ESP_BLE_MESH_PROVISIONER_SET_HEARTBEAT_FILTER_INFO_COMP_EVT
Provisioner set the heartbeat filter information completion event

ESP_BLE_MESH_PROVISIONER_RECV_HEARTBEAT_MESSAGE_EVT
Provisioner receive heartbeat message event

ESP_BLE_MESH_PROVISIONER_DIRECT_ERASE_SETTINGS_COMP_EVT
Provisioner directly erase settings completion event

ESP_BLE_MESH_PROVISIONER_OPEN_SETTINGS_WITH_INDEX_COMP_EVT
Provisioner open settings with index completion event

ESP_BLE_MESH_PROVISIONER_OPEN_SETTINGS_WITH_UID_COMP_EVT
Provisioner open settings with user id completion event

ESP_BLE_MESH_PROVISIONER_CLOSE_SETTINGS_WITH_INDEX_COMP_EVT
Provisioner close settings with index completion event

ESP_BLE_MESH_PROVISIONER_CLOSE_SETTINGS_WITH_UID_COMP_EVT
Provisioner close settings with user id completion event

ESP_BLE_MESH_PROVISIONER_DELETE_SETTINGS_WITH_INDEX_COMP_EVT
Provisioner delete settings with index completion event

ESP_BLE_MESH_PROVISIONER_DELETE_SETTINGS_WITH_UID_COMP_EVT
Provisioner delete settings with user id completion event

ESP_BLE_MESH_SET_FAST_PROV_INFO_COMP_EVT
Set fast provisioning information (e.g. unicast address range, net_idx, etc.) completion event

ESP_BLE_MESH_SET_FAST_PROV_ACTION_COMP_EVT
Set fast provisioning action completion event

ESP_BLE_MESH_HEARTBEAT_MESSAGE_RECV_EVT
Receive Heartbeat message event

ESP_BLE_MESH_LPN_ENABLE_COMP_EVT
Enable Low Power Node completion event

ESP_BLE_MESH_LPN_DISABLE_COMP_EVT
Disable Low Power Node completion event

ESP_BLE_MESH_LPN_POLL_COMP_EVT
Low Power Node send Friend Poll completion event

ESP_BLE_MESH_LPN_FRIENDSHIP_ESTABLISH_EVT
Low Power Node establishes friendship event

ESP_BLE_MESH_LPN_FRIENDSHIP_TERMINATE_EVT
Low Power Node terminates friendship event

ESP_BLE_MESH_FRIEND_FRIENDSHIP_ESTABLISH_EVT
Friend Node establishes friendship event

ESP_BLE_MESH_FRIEND_FRIENDSHIP_TERMINATE_EVT
Friend Node terminates friendship event

ESP_BLE_MESH_PROXY_CLIENT_RECV_ADV_PKT_EVT
Proxy Client receives Network ID advertising packet event

ESP_BLE_MESH_PROXY_CLIENT_CONNECTED_EVT
Proxy Client establishes connection successfully event

ESP_BLE_MESH_PROXY_CLIENT_DISCONNECTED_EVT
Proxy Client terminates connection successfully event

ESP_BLE_MESH_PROXY_CLIENT_RECV_FILTER_STATUS_EVT
Proxy Client receives Proxy Filter Status event

ESP_BLE_MESH_PROXY_CLIENT_CONNECT_COMP_EVT
Proxy Client connect completion event

ESP_BLE_MESH_PROXY_CLIENT_DISCONNECT_COMP_EVT
Proxy Client disconnect completion event

ESP_BLE_MESH_PROXY_CLIENT_SET_FILTER_TYPE_COMP_EVT
Proxy Client set filter type completion event

ESP_BLE_MESH_PROXY_CLIENT_ADD_FILTER_ADDR_COMP_EVT
Proxy Client add filter address completion event

ESP_BLE_MESH_PROXY_CLIENT_REMOVE_FILTER_ADDR_COMP_EVT
Proxy Client remove filter address completion event

ESP_BLE_MESH_MODEL_SUBSCRIBE_GROUP_ADDR_COMP_EVT
Local model subscribes group address completion event

ESP_BLE_MESH_MODEL_UNSUBSCRIBE_GROUP_ADDR_COMP_EVT
Local model unsubscribes group address completion event

ESP_BLE_MESH_DEINIT_MESH_COMP_EVT
De-initialize BLE Mesh stack completion event

ESP_BLE_MESH_PROV_EVT_MAX

enum [anonymous]

BLE Mesh server models related definitions.

This enum value is the flag of transition timer operation

Values:

ESP_BLE_MESH_SERVER_TRANS_TIMER_START
ESP_BLE_MESH_SERVER_FLAG_MAX

enum esp_ble_mesh_server_state_type_t

This enum value is the type of server model states

Values:

ESP_BLE_MESH_GENERIC_ONOFF_STATE
ESP_BLE_MESH_GENERIC_LEVEL_STATE
ESP_BLE_MESH_GENERIC_ONPOWERUP_STATE
ESP_BLE_MESH_GENERIC_POWER_ACTUAL_STATE
ESP_BLE_MESH_LIGHT_LIGHTNESS_ACTUAL_STATE
ESP_BLE_MESH_LIGHT_LIGHTNESS_LINEAR_STATE
ESP_BLE_MESH_LIGHT_CTL_LIGHTNESS_STATE
ESP_BLE_MESH_LIGHT_CTL_TEMP_DELTA_UV_STATE

```
ESP_BLE_MESH_LIGHT_HSL_STATE
ESP_BLE_MESH_LIGHT_HSL_LIGHTNESS_STATE
ESP_BLE_MESH_LIGHT_HSL_HUE_STATE
ESP_BLE_MESH_LIGHT_HSL_SATURATION_STATE
ESP_BLE_MESH_LIGHT_XYL_LIGHTNESS_STATE
ESP_BLE_MESH_LIGHT_LC_LIGHT_ONOFF_STATE
ESP_BLE_MESH_SERVER_MODEL_STATE_MAX
```

```
enum esp_ble_mesh_model_cb_event_t
```

Values:

```
ESP_BLE_MESH_MODEL_OPERATION_EVT
    User-defined models receive messages from peer devices (e.g. get, set, status, etc) event
ESP_BLE_MESH_MODEL_SEND_COMP_EVT
    User-defined models send messages completion event
ESP_BLE_MESH_MODEL_PUBLISH_COMP_EVT
    User-defined models publish messages completion event
ESP_BLE_MESH_CLIENT_MODEL_RECV_PUBLISH_MSG_EVT
    User-defined client models receive publish messages event
ESP_BLE_MESH_CLIENT_MODEL_SEND_TIMEOUT_EVT
    Timeout event for the user-defined client models that failed to receive response from peer server models
ESP_BLE_MESH_MODEL_PUBLISH_UPDATE_EVT
    When a model is configured to publish messages periodically, this event will occur during every publish
    period
ESP_BLE_MESH_SERVER_MODEL_UPDATE_STATE_COMP_EVT
    Server models update state value completion event
ESP_BLE_MESH_MODEL_EVT_MAX
```

ESP-BLE-MESH Core API Reference

This section contains ESP-BLE-MESH Core related APIs, which can be used to initialize ESP-BLE-MESH stack, provision, send/publish messages, etc.

This API reference covers six components:

- [ESP-BLE-MESH Stack Initialization](#)
- [Reading of Local Data Information](#)
- [Low Power Operation \(Updating\)](#)
- [Send/Publish Messages, add Local AppKey, etc.](#)
- [ESP-BLE-MESH Node/Provisioner Provisioning](#)
- [ESP-BLE-MESH GATT Proxy Server](#)

ESP-BLE-MESH Stack Initialization

Header File

- `bt/esp_ble_mesh/api/core/include/esp_ble_mesh_common_api.h`

Functions

esp_err_t **esp_ble_mesh_init** (*esp_ble_mesh_prov_t* *prov, *esp_ble_mesh_comp_t* *comp)

Initialize BLE Mesh module. This API initializes provisioning capabilities and composition data information.

Note After calling this API, the device needs to call `esp_ble_mesh_prov_enable()` to enable provisioning functionality again.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] prov: Pointer to the device provisioning capabilities. This pointer must remain valid during the lifetime of the BLE Mesh device.
- [in] comp: Pointer to the device composition data information. This pointer must remain valid during the lifetime of the BLE Mesh device.

esp_err_t **esp_ble_mesh_deinit** (*esp_ble_mesh_deinit_param_t* *param)

De-initialize BLE Mesh module.

Note This function shall be invoked after `esp_ble_mesh_client_model_deinit()`.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] param: Pointer to the structure of BLE Mesh deinit parameters.

Reading of Local Data Information

Header File

- [bt/esp_ble_mesh/api/core/include/esp_ble_mesh_local_data_operation_api.h](#)

Functions

int32_t **esp_ble_mesh_get_model_publish_period** (*esp_ble_mesh_model_t* *model)

Get the model publish period, the unit is ms.

Return Publish period value on success, 0 or (negative) error code from `errno.h` on failure.

Parameters

- [in] model: Model instance pointer.

uint16_t **esp_ble_mesh_get_primary_element_address** (void)

Get the address of the primary element.

Return Address of the primary element on success, or ESP_BLE_MESH_ADDR_UNASSIGNED on failure which means the device has not been provisioned.

uint16_t ***esp_ble_mesh_is_model_subscribed_to_group** (*esp_ble_mesh_model_t* *model, *uint16_t* group_addr)

Check if the model has subscribed to the given group address. Note: E.g., once a status message is received and the destination address is a group address, the model uses this API to check if it is successfully subscribed to the given group address.

Return Pointer to the group address within the Subscription List of the model on success, or NULL on failure which means the model has not subscribed to the given group address. Note: With the pointer to the group address returned, you can reset the group address to 0x0000 in order to unsubscribe the model from the group.

Parameters

- [in] model: Pointer to the model.
- [in] group_addr: Group address.

esp_ble_mesh_elem_t ***esp_ble_mesh_find_element** (*uint16_t* element_addr)

Find the BLE Mesh element pointer via the element address.

Return Pointer to the element on success, or NULL on failure.

Parameters

- [in] element_addr: Element address.

`uint8_t esp_ble_mesh_get_element_count` (void)

Get the number of elements that have been registered.

Return Number of elements.

`esp_ble_mesh_model_t *esp_ble_mesh_find_vendor_model` (const `esp_ble_mesh_elem_t` **element*, `uint16_t` *company_id*, `uint16_t` *model_id*)

Find the Vendor specific model with the given element, the company ID and the Vendor Model ID.

Return Pointer to the Vendor Model on success, or NULL on failure which means the Vendor Model is not found.

Parameters

- [in] *element*: Element to which the model belongs.
- [in] *company_id*: A 16-bit company identifier assigned by the Bluetooth SIG.
- [in] *model_id*: A 16-bit vendor-assigned model identifier.

`esp_ble_mesh_model_t *esp_ble_mesh_find_sig_model` (const `esp_ble_mesh_elem_t` **element*, `uint16_t` *model_id*)

Find the SIG model with the given element and Model id.

Return Pointer to the SIG Model on success, or NULL on failure which means the SIG Model is not found.

Parameters

- [in] *element*: Element to which the model belongs.
- [in] *model_id*: SIG model identifier.

`const esp_ble_mesh_comp_t *esp_ble_mesh_get_composition_data` (void)

Get the Composition data which has been registered.

Return Pointer to the Composition data on success, or NULL on failure which means the Composition data is not initialized.

`esp_err_t esp_ble_mesh_model_subscribe_group_addr` (`uint16_t` *element_addr*, `uint16_t` *company_id*, `uint16_t` *model_id*, `uint16_t` *group_addr*)

A local model of node or Provisioner subscribes a group address.

Note This function shall not be invoked before node is provisioned or Provisioner is enabled.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] *element_addr*: Unicast address of the element to which the model belongs.
- [in] *company_id*: A 16-bit company identifier.
- [in] *model_id*: A 16-bit model identifier.
- [in] *group_addr*: The group address to be subscribed.

`esp_err_t esp_ble_mesh_model_unsubscribe_group_addr` (`uint16_t` *element_addr*, `uint16_t` *company_id*, `uint16_t` *model_id*, `uint16_t` *group_addr*)

A local model of node or Provisioner unsubscribes a group address.

Note This function shall not be invoked before node is provisioned or Provisioner is enabled.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] *element_addr*: Unicast address of the element to which the model belongs.
- [in] *company_id*: A 16-bit company identifier.
- [in] *model_id*: A 16-bit model identifier.
- [in] *group_addr*: The subscribed group address.

`const uint8_t *esp_ble_mesh_node_get_local_net_key` (`uint16_t` *net_idx*)

This function is called by Node to get the local NetKey.

Return NetKey on success, or NULL on failure.

Parameters

- [in] *net_idx*: NetKey index.

const uint8_t ***esp_ble_mesh_node_get_local_app_key** (uint16_t *app_idx*)

This function is called by Node to get the local AppKey.

Return AppKey on success, or NULL on failure.

Parameters

- [in] *app_idx*: AppKey index.

esp_err_t **esp_ble_mesh_node_add_local_net_key** (**const** uint8_t *net_key*[16], uint16_t *net_idx*)

This function is called by Node to add a local NetKey.

Note This function can only be called after the device is provisioned.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] *net_key*: NetKey to be added.
- [in] *net_idx*: NetKey Index.

esp_err_t **esp_ble_mesh_node_add_local_app_key** (**const** uint8_t *app_key*[16], uint16_t *net_idx*, uint16_t *app_idx*)

This function is called by Node to add a local AppKey.

Note The *net_idx* must be an existing one. This function can only be called after the device is provisioned.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] *app_key*: AppKey to be added.
- [in] *net_idx*: NetKey Index.
- [in] *app_idx*: AppKey Index.

esp_err_t **esp_ble_mesh_node_bind_app_key_to_local_model** (uint16_t *element_addr*,
uint16_t *company_id*,
uint16_t *model_id*, uint16_t *app_idx*)

This function is called by Node to bind AppKey to model locally.

Note If going to bind *app_key* with local vendor model, the *company_id* shall be set to 0xFFFF. This function can only be called after the device is provisioned.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] *element_addr*: Node local element address
- [in] *company_id*: Node local company id
- [in] *model_id*: Node local model id
- [in] *app_idx*: Node local appkey index

Low Power Operation (Updating)

Header File

- [bt/esp_ble_mesh/api/core/include/esp_ble_mesh_low_power_api.h](#)

Functions

esp_err_t **esp_ble_mesh_lpn_enable** (void)

Enable BLE Mesh device LPN functionality.

Note This API enables LPN functionality. Once called, the proper Friend Request will be sent.

Return ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_lpn_disable** (bool *force*)

Disable BLE Mesh device LPN functionality.

Return ESP_OK on success or error code otherwise.

Parameters

- `[in] force`: when disabling LPN functionality, use this flag to indicate whether directly clear corresponding information or just send friend clear to disable it if friendship has already been established.

`esp_err_t esp_ble_mesh_lpn_poll` (void)

LPN tries to poll messages from the Friend Node.

Note The Friend Poll message is sent by a Low Power node to ask the Friend node to send a message that it has stored for the Low Power node. Users can call this API to send Friend Poll message manually. If this API is not invoked, the bottom layer of the Low Power node will send Friend Poll before the PollTimeout timer expires. If the corresponding Friend Update is received and MD is set to 0, which means there are no messages for the Low Power node, then the Low Power node will stop scanning.

Return ESP_OK on success or error code otherwise.

Send/Publish Messages, add Local AppKey, etc.

Header File

- [bt/esp_ble_mesh/api/core/include/esp_ble_mesh_networking_api.h](#)

Functions

`esp_err_t esp_ble_mesh_register_custom_model_callback` (*esp_ble_mesh_model_cb_t callback*)

Register BLE Mesh callback for user-defined models' operations. This callback can report the following events generated for the user-defined models:

- Call back the messages received by user-defined client and server models to the application layer;
- If users call `esp_ble_mesh_server/client_model_send`, this callback notifies the application layer of the `send_complete` event;
- If user-defined client model sends a message that requires response, and the response message is received after the timer expires, the response message will be reported to the application layer as published by a peer device;
- If the user-defined client model fails to receive the response message during a specified period of time, a timeout event will be reported to the application layer.

Note The client models (i.e. Config Client model, Health Client model, Generic Client models, Sensor Client model, Scene Client model and Lighting Client models) that have been realized internally have their specific register functions. For example, `esp_ble_mesh_register_config_client_callback` is the register function for Config Client Model.

Return ESP_OK on success or error code otherwise.

Parameters

- `[in] callback`: Pointer to the callback function.

`esp_err_t esp_ble_mesh_model_msg_opcode_init` (uint8_t *data, uint32_t opcode)

Add the message opcode to the beginning of the model message before sending or publishing the model message.

Note This API is only used to set the opcode of the message.

Return ESP_OK on success or error code otherwise.

Parameters

- `[in] data`: Pointer to the message data.
- `[in] opcode`: The message opcode.

`esp_err_t esp_ble_mesh_client_model_init` (*esp_ble_mesh_model_t *model*)

Initialize the user-defined client model. All user-defined client models shall call this function to initialize the client model internal data. Note: Before calling this API, the `op_pair_size` and `op_pair` variables within the `user_data` (defined using `esp_ble_mesh_client_t`) of the client model need to be initialized.

Return ESP_OK on success or error code otherwise.

Parameters

- `[in] model`: BLE Mesh Client model to which the message belongs.

esp_err_t **esp_ble_mesh_client_model_deinit** (*esp_ble_mesh_model_t* *model)

De-initialize the user-defined client model.

Note This function shall be invoked before `esp_ble_mesh_deinit()` is called.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] model: Pointer of the Client model.

esp_err_t **esp_ble_mesh_server_model_send_msg** (*esp_ble_mesh_model_t* *model,
esp_ble_mesh_msg_ctx_t *ctx, uint32_t
opcode, uint16_t length, uint8_t *data)

Send server model messages (such as server model status messages).

Return ESP_OK on success or error code otherwise.

Parameters

- [in] model: BLE Mesh Server Model to which the message belongs.
- [in] ctx: Message context, includes keys, TTL, etc.
- [in] opcode: Message opcode.
- [in] length: Message length (exclude the message opcode).
- [in] data: Parameters of Access Payload (exclude the message opcode) to be sent.

esp_err_t **esp_ble_mesh_client_model_send_msg** (*esp_ble_mesh_model_t* *model,
esp_ble_mesh_msg_ctx_t *ctx, uint32_t
opcode, uint16_t length, uint8_t *data,
int32_t msg_timeout, bool need_rsp,
esp_ble_mesh_dev_role_t device_role)

Send client model message (such as model get, set, etc).

Return ESP_OK on success or error code otherwise.

Parameters

- [in] model: BLE Mesh Client Model to which the message belongs.
- [in] ctx: Message context, includes keys, TTL, etc.
- [in] opcode: Message opcode.
- [in] length: Message length (exclude the message opcode).
- [in] data: Parameters of the Access Payload (exclude the message opcode) to be sent.
- [in] msg_timeout: Time to get response to the message (in milliseconds).
- [in] need_rsp: TRUE if the opcode requires the peer device to reply, FALSE otherwise.
- [in] device_role: Role of the device (Node/Provisioner) that sends the message.

esp_err_t **esp_ble_mesh_model_publish** (*esp_ble_mesh_model_t* *model, uint32_t opcode, uint16_t
length, uint8_t *data, *esp_ble_mesh_dev_role_t* de-
vice_role)

Send a model publication message.

Note Before calling this function, the user needs to ensure that the model publication message (*esp_ble_mesh_model_pub_t:msg*) contains a valid message to be sent. And if users want to update the publishing message, this API should be called in ESP_BLE_MESH_MODEL_PUBLISH_UPDATE_EVT with the message updated.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] model: Mesh (client) Model publishing the message.
- [in] opcode: Message opcode.
- [in] length: Message length (exclude the message opcode).
- [in] data: Parameters of the Access Payload (exclude the message opcode) to be sent.
- [in] device_role: Role of the device (node/provisioner) publishing the message of the type `esp_ble_mesh_dev_role_t`.

esp_err_t **esp_ble_mesh_server_model_update_state** (*esp_ble_mesh_model_t* *model,
esp_ble_mesh_server_state_type_t type,
esp_ble_mesh_server_state_value_t
*value)

Update a server model state value. If the model publication state is set properly (e.g. publish address is set to a valid address), it will publish corresponding status message.

Note Currently this API is used to update bound state value, not for all server model states.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] model: Server model which is going to update the state.
- [in] type: Server model state type.
- [in] value: Server model state value.

esp_err_t **esp_ble_mesh_node_local_reset** (void)

Reset the provisioning procedure of the local BLE Mesh node.

Note All provisioning information in this node will be deleted and the node needs to be reprovisioned. The API function `esp_ble_mesh_node_prov_enable()` needs to be called to start a new provisioning procedure.

Return ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_provisioner_set_node_name** (uint16_t *index*, const char **name*)

This function is called to set the node (provisioned device) name.

Note *index* is obtained from the parameters of ESP_BLE_MESH_PROVISIONER_PROV_COMPLETE_EVT.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] index: Index of the node in the node queue.
- [in] name: Name (end by '\0') to be set for the node.

const char ***esp_ble_mesh_provisioner_get_node_name** (uint16_t *index*)

This function is called to get the node (provisioned device) name.

Note *index* is obtained from the parameters of ESP_BLE_MESH_PROVISIONER_PROV_COMPLETE_EVT.

Return Node name on success, or NULL on failure.

Parameters

- [in] index: Index of the node in the node queue.

uint16_t **esp_ble_mesh_provisioner_get_node_index** (const char **name*)

This function is called to get the node (provisioned device) index.

Return Node index on success, or an invalid value (0xFFFF) on failure.

Parameters

- [in] name: Name of the node (end by '\0').

esp_err_t **esp_ble_mesh_provisioner_store_node_comp_data** (uint16_t *unicast_addr*,
uint8_t **data*, uint16_t
length)

This function is called to store the Composition Data of the node.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] unicast_addr: Element address of the node
- [in] data: Pointer of Composition Data
- [in] length: Length of Composition Data

esp_ble_mesh_node_t ***esp_ble_mesh_provisioner_get_node_with_uuid** (const uint8_t
uuid[16])

This function is called to get the provisioned node information with the node device uuid.

Return Pointer of the node info struct or NULL on failure.

Parameters

- [in] uuid: Device UUID of the node

esp_ble_mesh_node_t ***esp_ble_mesh_provisioner_get_node_with_addr** (uint16_t *uni-*
cast_addr)

This function is called to get the provisioned node information with the node unicast address.

Return Pointer of the node info struct or NULL on failure.

Parameters

- [in] unicast_addr: Unicast address of the node

esp_ble_mesh_node_t ***esp_ble_mesh_provisioner_get_node_with_name**(const char *name)

This function is called to get the provisioned node information with the node name.

Return Pointer of the node info struct or NULL on failure.

Parameters

- [in] name: Name of the node (end by '\0').

uint16_t **esp_ble_mesh_provisioner_get_prov_node_count**(void)

This function is called by Provisioner to get provisioned node count.

Return Number of the provisioned nodes.

const *esp_ble_mesh_node_t* ****esp_ble_mesh_provisioner_get_node_table_entry**(void)

This function is called by Provisioner to get the entry of the node table.

Note After invoking the function to get the entry of nodes, users can use the “for” loop combined with the macro CONFIG_BLE_MESH_MAX_PROV_NODES to get each node’s information. Before trying to read the node’s information, users need to check if the node exists, i.e. if the *(*esp_ble_mesh_node_t* **node) is NULL. For example: `` const *esp_ble_mesh_node_t* **entry = esp_ble_mesh_provisioner_get_node_table_entry(); for (int i = 0; i < CONFIG_BLE_MESH_MAX_PROV_NODES; i++) { const *esp_ble_mesh_node_t* *node = entry[i]; if (node) { …… } } ``

Return Pointer to the start of the node table.

esp_err_t **esp_ble_mesh_provisioner_delete_node_with_uuid**(const uint8_t uuid[16])

This function is called to delete the provisioned node information with the node device uuid.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] uuid: Device UUID of the node

esp_err_t **esp_ble_mesh_provisioner_delete_node_with_addr**(uint16_t unicast_addr)

This function is called to delete the provisioned node information with the node unicast address.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] unicast_addr: Unicast address of the node

esp_err_t **esp_ble_mesh_provisioner_add_local_app_key**(const uint8_t app_key[16], uint16_t net_idx, uint16_t app_idx)

This function is called to add a local AppKey for Provisioner.

Note app_key: If set to NULL, app_key will be generated internally. net_idx: Should be an existing one. app_idx: If it is going to be generated internally, it should be set to 0xFFFF, and the new app_idx will be reported via an event.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] app_key: The app key to be set for the local BLE Mesh stack.
- [in] net_idx: The network key index.
- [in] app_idx: The app key index.

esp_err_t **esp_ble_mesh_provisioner_update_local_app_key**(const uint8_t app_key[16], uint16_t net_idx, uint16_t app_idx)

This function is used to update a local AppKey for Provisioner.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] app_key: Value of the AppKey.
- [in] net_idx: Corresponding NetKey Index.
- [in] app_idx: The AppKey Index

```
const uint8_t *esp_ble_mesh_provisioner_get_local_app_key (uint16_t net_idx, uint16_t app_idx)
```

This function is called by Provisioner to get the local app key value.

Return App key on success, or NULL on failure.

Parameters

- [in] *net_idx*: Network key index.
- [in] *app_idx*: Application key index.

```
esp_err_t esp_ble_mesh_provisioner_bind_app_key_to_local_model (uint16_t element_addr,  
uint16_t app_idx,  
uint16_t model_id,  
uint16_t company_id)
```

This function is called by Provisioner to bind own model with proper app key.

Note *company_id*: If going to bind *app_key* with local vendor model, *company_id* should be set to 0xFFFF.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] *element_addr*: Provisioner local element address
- [in] *app_idx*: Provisioner local appkey index
- [in] *model_id*: Provisioner local model id
- [in] *company_id*: Provisioner local company id

```
esp_err_t esp_ble_mesh_provisioner_add_local_net_key (const uint8_t net_key[16],  
uint16_t net_idx)
```

This function is called by Provisioner to add local network key.

Note *net_key*: If set to NULL, *net_key* will be generated internally. *net_idx*: If it is going to be generated internally, it should be set to 0xFFFF, and the new *net_idx* will be reported via an event.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] *net_key*: The network key to be added to the Provisioner local BLE Mesh stack.
- [in] *net_idx*: The network key index.

```
esp_err_t esp_ble_mesh_provisioner_update_local_net_key (const uint8_t net_key[16],  
uint16_t net_idx)
```

This function is called by Provisioner to update a local network key.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] *net_key*: Value of the NetKey.
- [in] *net_idx*: The NetKey Index.

```
const uint8_t *esp_ble_mesh_provisioner_get_local_net_key (uint16_t net_idx)
```

This function is called by Provisioner to get the local network key value.

Return Network key on success, or NULL on failure.

Parameters

- [in] *net_idx*: Network key index.

```
esp_err_t esp_ble_mesh_provisioner_recv_heartbeat (bool enable)
```

This function is called by Provisioner to enable or disable receiving heartbeat messages.

Note If enabling receiving heartbeat message successfully, the filter will be an empty rejectlist by default, which means all heartbeat messages received by the Provisioner will be reported to the application layer.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] *enable*: Enable or disable receiving heartbeat messages.

```
esp_err_t esp_ble_mesh_provisioner_set_heartbeat_filter_type (uint8_t type)
```

This function is called by Provisioner to set the heartbeat filter type.

- Note** 1. If the filter type is not the same with the current value, then all the filter entries will be cleaned.
1. If the previous type is rejectlist, and changed to acceptlist, then the filter will be an empty acceptlist, which means no heartbeat messages will be reported. Users need to add SRC or DST into the filter entry, then heartbeat messages from the SRC or to the DST will be reported.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] type: Heartbeat filter type (acceptlist or rejectlist).

esp_err_t **esp_ble_mesh_provisioner_set_heartbeat_filter_info** (uint8_t *op*,
esp_ble_mesh_heartbeat_filter_info_t **info*)

This function is called by Provisioner to add or remove a heartbeat filter entry.

1. If the operation is “REMOVE” , the “hb_src” can be set to the SRC (can only be a unicast address) of heartbeat messages, and the “hb_dst” can be set to the DST (unicast address or group address), at least one of them needs to be set.
 - The filter entry with the same SRC or DST will be removed.

Note 1. If the operation is “ADD” , the “hb_src” can be set to the SRC (can only be a unicast address) of heartbeat messages, and the “hb_dst” can be set to the DST (unicast address or group address), at least one of them needs to be set.

- If only one of them is set, the filter entry will only use the configured SRC or DST to filter heartbeat messages.
- If both of them are set, the SRC and DST will both be used to decide if a heartbeat message will be handled.
- If SRC or DST already exists in some filter entry, then the corresponding entry will be cleaned firstly, then a new entry will be allocated to store the information.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] op: Add or REMOVE
- [in] info: Heartbeat filter entry information, including: hb_src - Heartbeat source address; hb_dst - Heartbeat destination address;

esp_err_t **esp_ble_mesh_provisioner_direct_erase_settings** (void)

This function is called by Provisioner to directly erase the mesh information from nvs namespace.

Note This function can be invoked when the mesh stack is not initialized or has been de-initialized.

Return ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_provisioner_open_settings_with_index** (uint8_t *index*)

This function is called by Provisioner to open a nvs namespace for storing mesh information.

Note Before open another nvs namespace, the previously opened nvs namespace must be closed firstly.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] index: Settings index.

esp_err_t **esp_ble_mesh_provisioner_open_settings_with_uid** (const char **uid*)

This function is called by Provisioner to open a nvs namespace for storing mesh information.

Note Before open another nvs namespace, the previously opened nvs namespace must be closed firstly.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] uid: Settings user id.

esp_err_t **esp_ble_mesh_provisioner_close_settings_with_index** (uint8_t *index*, bool *erase*)

This function is called by Provisioner to close a nvs namespace which is opened previously for storing mesh information.

Note 1. Before closing the nvs namespace, it must be open.

1. When the function is invoked, the Provisioner functionality will be disabled firstly, and: a) If the “erase” flag is set to false, the mesh information will be cleaned (e.g. removing NetKey, AppKey,

- nodes, etc) from the mesh stack. b) If the “erase” flag is set to true, the mesh information stored in the nvs namespace will also be erased besides been cleaned from the mesh stack.
2. If Provisioner tries to work properly again, we can invoke the open function to open a new nvs namespace or a previously added one, and restore the mesh information from it if not erased.
 3. The working process shall be as following: a) Open settings A b) Start to provision and control nodes c) Close settings A d) Open settings B e) Start to provision and control other nodes f) Close settings B g) ……

Return ESP_OK on success or error code otherwise.

Parameters

- [in] index: Settings index.
- [in] erase: Indicate if erasing mesh information.

esp_err_t **esp_ble_mesh_provisioner_close_settings_with_uid**(const char *uid, bool erase)

This function is called by Provisioner to close a nvs namespace which is opened previously for storing mesh information.

Note 1. Before closing the nvs namespace, it must be open.

1. When the function is invoked, the Provisioner functionality will be disabled firstly, and: a) If the “erase” flag is set to false, the mesh information will be cleaned (e.g. removing NetKey, AppKey, nodes, etc) from the mesh stack. b) If the “erase” flag is set to true, the mesh information stored in the nvs namespace will also be erased besides been cleaned from the mesh stack.
2. If Provisioner tries to work properly again, we can invoke the open function to open a new nvs namespace or a previously added one, and restore the mesh information from it if not erased.
3. The working process shall be as following: a) Open settings A b) Start to provision and control nodes c) Close settings A d) Open settings B e) Start to provision and control other nodes f) Close settings B g) ……

Return ESP_OK on success or error code otherwise.

Parameters

- [in] uid: Settings user id.
- [in] erase: Indicate if erasing mesh information.

esp_err_t **esp_ble_mesh_provisioner_delete_settings_with_index**(uint8_t index)

This function is called by Provisioner to erase the mesh information and settings user id from a nvs namespace.

Note When this function is called, the nvs namespace must not be open. This function is used to erase the mesh information and settings user id which are not used currently.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] index: Settings index.

esp_err_t **esp_ble_mesh_provisioner_delete_settings_with_uid**(const char *uid)

This function is called by Provisioner to erase the mesh information and settings user id from a nvs namespace.

Note When this function is called, the nvs namespace must not be open. This function is used to erase the mesh information and settings user id which are not used currently.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] uid: Settings user id.

const char ***esp_ble_mesh_provisioner_get_settings_uid**(uint8_t index)

This function is called by Provisioner to get settings user id.

Return Setting user id on success or NULL on failure.

Parameters

- [in] index: Settings index.

uint8_t **esp_ble_mesh_provisioner_get_settings_index**(const char *uid)

This function is called by Provisioner to get settings index.

Return Settings index.

Parameters

- [in] uid: Settings user id.

`uint8_t esp_ble_mesh_provisioner_get_free_settings_count` (void)

This function is called by Provisioner to get the number of free settings user id.

Return Number of free settings user id.

`const uint8_t *esp_ble_mesh_get_fast_prov_app_key` (uint16_t *net_idx*, uint16_t *app_idx*)

This function is called to get fast provisioning application key.

Return Application key on success, or NULL on failure.

Parameters

- [in] *net_idx*: Network key index.
- [in] *app_idx*: Application key index.

Type Definitions

```
typedef void (*esp_ble_mesh_model_cb_t) (esp_ble_mesh_model_cb_event_t          event,  
                                         esp_ble_mesh_model_cb_param_t *param)  
: event, event code of user-defined model events; param, parameters of user-defined model events
```

ESP-BLE-MESH Node/Provisioner Provisioning

Header File

- [bt/esp_ble_mesh/api/core/include/esp_ble_mesh_provisioning_api.h](#)

Functions

`esp_err_t esp_ble_mesh_register_prov_callback` (*esp_ble_mesh_prov_cb_t* *callback*)

Register BLE Mesh provisioning callback.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] *callback*: Pointer to the callback function.

`bool esp_ble_mesh_node_is_provisioned` (void)

Check if a device has been provisioned.

Return TRUE if the device is provisioned, FALSE if the device is unprovisioned.

`esp_err_t esp_ble_mesh_node_prov_enable` (*esp_ble_mesh_prov_bearer_t* *bearers*)

Enable specific provisioning bearers to get the device ready for provisioning.

Note PB-ADV: send unprovisioned device beacon. PB-GATT: send connectable advertising packets.

Return ESP_OK on success or error code otherwise.

Parameters

- *bearers*: Bit-wise OR of provisioning bearers.

`esp_err_t esp_ble_mesh_node_prov_disable` (*esp_ble_mesh_prov_bearer_t* *bearers*)

Disable specific provisioning bearers to make a device inaccessible for provisioning.

Return ESP_OK on success or error code otherwise.

Parameters

- *bearers*: Bit-wise OR of provisioning bearers.

`esp_err_t esp_ble_mesh_node_set_oob_pub_key` (uint8_t *pub_key_x*[32], uint8_t *pub_key_y*[32],
 uint8_t *private_key*[32])

Unprovisioned device set own oob public key & private key pair.

Note In order to avoid suffering brute-forcing attack (CVE-2020-26559). The Bluetooth SIG recommends that potentially vulnerable mesh provisioners use an out-of-band mechanism to exchange the public keys. So as an unprovisioned device, it should use this function to input the Public Key exchanged through the out-of-band mechanism.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] *pub_key_x*: Unprovisioned device's Public Key X

- [in] `pub_key_y`: Unprovisioned device' s Public Key Y
- [in] `private_key`: Unprovisioned device' s Private Key

esp_err_t **esp_ble_mesh_node_input_number** (uint32_t *number*)

Provide provisioning input OOB number.

Note This is intended to be called if the user has received `ESP_BLE_MESH_NODE_PROV_INPUT_EVT` with `ESP_BLE_MESH_ENTER_NUMBER` as the action.

Return `ESP_OK` on success or error code otherwise.

Parameters

- [in] `number`: Number input by device.

esp_err_t **esp_ble_mesh_node_input_string** (const char **string*)

Provide provisioning input OOB string.

Note This is intended to be called if the user has received `ESP_BLE_MESH_NODE_PROV_INPUT_EVT` with `ESP_BLE_MESH_ENTER_STRING` as the action.

Return `ESP_OK` on success or error code otherwise.

Parameters

- [in] `string`: String input by device.

esp_err_t **esp_ble_mesh_set_unprovisioned_device_name** (const char **name*)

Using this function, an unprovisioned device can set its own device name, which will be broadcasted in its advertising data.

Note This API applicable to PB-GATT mode only by setting the name to the scan response data, it doesn' t apply to PB-ADV mode.

Return `ESP_OK` on success or error code otherwise.

Parameters

- [in] `name`: Unprovisioned device name

esp_err_t **esp_ble_mesh_provisioner_read_oob_pub_key** (uint8_t *link_idx*, uint8_t *pub_key_x*[32], uint8_t *pub_key_y*[32])

Provisioner inputs unprovisioned device' s oob public key.

Note In order to avoid suffering brute-forcing attack (CVE-2020-26559). The Bluetooth SIG recommends that potentially vulnerable mesh provisioners use an out-of-band mechanism to exchange the public keys.

Return `ESP_OK` on success or error code otherwise.

Parameters

- [in] `link_idx`: The provisioning link index
- [in] `pub_key_x`: Unprovisioned device' s Public Key X
- [in] `pub_key_y`: Unprovisioned device' s Public Key Y

esp_err_t **esp_ble_mesh_provisioner_input_string** (const char **string*, uint8_t *link_idx*)

Provide provisioning input OOB string.

This is intended to be called after the *esp_ble_mesh_prov_t* `prov_input_num` callback has been called with `ESP_BLE_MESH_ENTER_STRING` as the action.

Return `ESP_OK` on success or error code otherwise.

Parameters

- [in] `string`: String input by Provisioner.
- [in] `link_idx`: The provisioning link index.

esp_err_t **esp_ble_mesh_provisioner_input_number** (uint32_t *number*, uint8_t *link_idx*)

Provide provisioning input OOB number.

This is intended to be called after the *esp_ble_mesh_prov_t* `prov_input_num` callback has been called with `ESP_BLE_MESH_ENTER_NUMBER` as the action.

Return `ESP_OK` on success or error code otherwise.

Parameters

- [in] `number`: Number input by Provisioner.
- [in] `link_idx`: The provisioning link index.

esp_err_t **esp_ble_mesh_provisioner_prov_enable** (*esp_ble_mesh_prov_bearer_t* bearers)
Enable one or more provisioning bearers.

Note PB-ADV: Enable BLE scan. PB-GATT: Initialize corresponding BLE Mesh Proxy info.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] bearers: Bit-wise OR of provisioning bearers.

esp_err_t **esp_ble_mesh_provisioner_prov_disable** (*esp_ble_mesh_prov_bearer_t* bearers)
Disable one or more provisioning bearers.

Note PB-ADV: Disable BLE scan. PB-GATT: Break any existing BLE Mesh Provisioning connections.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] bearers: Bit-wise OR of provisioning bearers.

esp_err_t **esp_ble_mesh_provisioner_add_unprov_dev** (*esp_ble_mesh_unprov_dev_add_t* *add_dev,
esp_ble_mesh_dev_add_flag_t flags)

Add unprovisioned device info to the unprov_dev queue.

Return ESP_OK on success or error code otherwise.

Note : 1. Currently address type only supports public address and static random address.

1. If device UUID and/or device address as well as address type already exist in the device queue, but the bearer is different from the existing one, add operation will also be successful and it will update the provision bearer supported by the device.
2. For example, if the Provisioner wants to add an unprovisioned device info before receiving its unprovisioned device beacon or Mesh Provisioning advertising packets, the Provisioner can use this API to add the device info with each one or both of device UUID and device address added. When the Provisioner gets the device's advertising packets, it will start provisioning the device internally.
 - In this situation, the Provisioner can set bearers with each one or both of ESP_BLE_MESH_PROV_ADV and ESP_BLE_MESH_PROV_GATT enabled, and cannot set flags with ADD_DEV_START_PROV_NOW_FLAG enabled.
3. Another example is when the Provisioner receives the unprovisioned device's beacon or Mesh Provisioning advertising packets, the advertising packets will be reported on to the application layer using the callback registered by the function esp_ble_mesh_register_prov_callback. And in the callback, the Provisioner can call this API to start provisioning the device.
 - If the Provisioner uses PB-ADV to provision, either one or both of device UUID and device address can be added, bearers shall be set with ESP_BLE_MESH_PROV_ADV enabled and the flags shall be set with ADD_DEV_START_PROV_NOW_FLAG enabled.
 - If the Provisioner uses PB-GATT to provision, both the device UUID and device address need to be added, bearers shall be set with ESP_BLE_MESH_PROV_GATT enabled, and the flags shall be set with ADD_DEV_START_PROV_NOW_FLAG enabled.
 - If the Provisioner just wants to store the unprovisioned device info when receiving its advertising packets and start to provision it the next time (e.g. after receiving its advertising packets again), then it can add the device info with either one or both of device UUID and device address included. Bearers can be set with either one or both of ESP_BLE_MESH_PROV_ADV and ESP_BLE_MESH_PROV_GATT enabled (recommend to enable the bearer which will receive its advertising packets, because if the other bearer is enabled, the Provisioner is not aware if the device supports the bearer), and flags cannot be set with ADD_DEV_START_PROV_NOW_FLAG enabled.
 - Note: ESP_BLE_MESH_PROV_ADV, ESP_BLE_MESH_PROV_GATT and ADD_DEV_START_PROV_NOW_FLAG can not be enabled at the same time.

Parameters

- [in] add_dev: Pointer to a struct containing the device information
- [in] flags: Flags indicate several operations on the device information
 - Remove device information from queue after device has been provisioned (BIT0)
 - Start provisioning immediately after device is added to queue (BIT1)
 - Device can be removed if device queue is full (BIT2)

AuthValues selected using a cryptographically secure random or pseudorandom number generator and having the maximum permitted entropy (128-bits) will be most difficult to brute-force. AuthValues with reduced entropy or generated in a predictable manner will not grant the same level of protection against this vulnerability. Selecting a new AuthValue with each provisioning attempt can also make it more difficult to launch a brute-force attack by requiring the attacker to restart the search with each provisioning attempt (CVE-2020-26556).

Note The Bluetooth SIG recommends that mesh implementations enforce a randomly selected AuthValue using all of the available bits, where permitted by the implementation. A large entropy helps ensure that a brute-force of the AuthValue, even a static AuthValue, cannot normally be completed in a reasonable time (CVE-2020-26557).

Return ESP_OK on success or error code otherwise.

Parameters

- [in] *value*: Pointer to the static oob value.
- [in] *length*: Length of the static oob value.

esp_err_t **esp_ble_mesh_provisioner_set_primary_elem_addr** (uint16_t *addr*)

This function is called by Provisioner to set own Primary element address.

Note This API must be invoked when BLE Mesh initialization is completed successfully, and can be invoked before Provisioner functionality is enabled. Once this API is invoked successfully, the *prov_unicast_addr* value in the struct *esp_ble_mesh_prov_t* will be ignored, and Provisioner will use this address as its own primary element address. And if the unicast address going to assigned for the next unprovisioned device is smaller than the input address + element number of Provisioner, then the address for the next unprovisioned device will be recalculated internally.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] *addr*: Unicast address of the Primary element of Provisioner.

esp_err_t **esp_ble_mesh_set_fast_prov_info** (*esp_ble_mesh_fast_prov_info_t* **fast_prov_info*)

This function is called to set provisioning data information before starting fast provisioning.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] *fast_prov_info*: Pointer to a struct containing unicast address range, *net_idx*, etc.

esp_err_t **esp_ble_mesh_set_fast_prov_action** (*esp_ble_mesh_fast_prov_action_t* *action*)

This function is called to start/suspend/exit fast provisioning.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] *action*: fast provisioning action (i.e. enter, suspend, exit).

Type Definitions

```
typedef void (*esp_ble_mesh_prov_cb_t) (esp_ble_mesh_prov_cb_event_t          event,
                                        esp_ble_mesh_prov_cb_param_t *param)
: event, event code of provisioning events; param, parameters of provisioning events
typedef void (*esp_ble_mesh_prov_adv_cb_t) (const esp_ble_mesh_bd_addr_t    addr,
                                           const esp_ble_mesh_addr_type_t   addr_type,
                                           const uint8_t                    adv_type,
                                           const uint8_t                    *dev_uuid,
                                           uint16_t                        oob_info,
                                           esp_ble_mesh_prov_bearer_t        bearer)
```

Callback for Provisioner that received advertising packets from unprovisioned devices which are not in the unprovisioned device queue.

Report on the unprovisioned device beacon and mesh provisioning service adv data to application.

Parameters

- [in] *addr*: Pointer to the unprovisioned device address.
- [in] *addr_type*: Unprovisioned device address type.
- [in] *adv_type*: Adv packet type (ADV_IND or ADV_NONCONN_IND).
- [in] *dev_uuid*: Unprovisioned device UUID pointer.
- [in] *oob_info*: OOB information of the unprovisioned device.

- [in] bearer: Adv packet received from PB-GATT or PB-ADV bearer.

ESP-BLE-MESH GATT Proxy Server

Header File

- [bt/esp_ble_mesh/api/core/include/esp_ble_mesh_proxy_api.h](#)

Functions

esp_err_t **esp_ble_mesh_proxy_identity_enable** (void)

Enable advertising with Node Identity.

Note This API requires that GATT Proxy support be enabled. Once called, each subnet starts advertising using Node Identity for the next 60 seconds, and after 60s Network ID will be advertised. Under normal conditions, the BLE Mesh Proxy Node Identity and Network ID advertising will be enabled automatically by BLE Mesh stack after the device is provisioned.

Return ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_proxy_gatt_enable** (void)

Enable BLE Mesh GATT Proxy Service.

Return ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_proxy_gatt_disable** (void)

Disconnect the BLE Mesh GATT Proxy connection if there is any, and disable the BLE Mesh GATT Proxy Service.

Return ESP_OK on success or error code otherwise.

esp_err_t **esp_ble_mesh_proxy_client_connect** (*esp_ble_mesh_bd_addr_t* *addr*,
esp_ble_mesh_addr_type_t *addr_type*, *uint16_t*
net_idx)

Proxy Client creates a connection with the Proxy Server.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] *addr*: Device address of the Proxy Server.
- [in] *addr_type*: Device address type(public or static random).
- [in] *net_idx*: NetKey Index related with Network ID in the Mesh Proxy advertising packet.

esp_err_t **esp_ble_mesh_proxy_client_disconnect** (*uint8_t* *conn_handle*)

Proxy Client terminates a connection with the Proxy Server.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] *conn_handle*: Proxy connection handle.

esp_err_t **esp_ble_mesh_proxy_client_set_filter_type** (*uint8_t* *conn_handle*,
uint16_t *net_idx*,
esp_ble_mesh_proxy_filter_type_t
filter_type)

Proxy Client sets the filter type of the Proxy Server.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] *conn_handle*: Proxy connection handle.
- [in] *net_idx*: Corresponding NetKey Index.
- [in] *filter_type*: whitelist or blacklist.

esp_err_t **esp_ble_mesh_proxy_client_add_filter_addr** (*uint8_t* *conn_handle*, *uint16_t*
net_idx, *uint16_t* **addr*, *uint16_t*
addr_num)

Proxy Client adds address to the Proxy Server filter list.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] `conn_handle`: Proxy connection handle.
- [in] `net_idx`: Corresponding NetKey Index.
- [in] `addr`: Pointer to the filter address.
- [in] `addr_num`: Number of the filter address.

`esp_err_t esp_ble_mesh_proxy_client_remove_filter_addr` (`uint8_t conn_handle`, `uint16_t net_idx`, `uint16_t *addr`, `uint16_t addr_num`)

Proxy Client removes address from the Proxy Server filter list.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] `conn_handle`: Proxy connection handle.
- [in] `net_idx`: Corresponding NetKey Index.
- [in] `addr`: Pointer to the filter address.
- [in] `addr_num`: Number of the filter address.

ESP-BLE-MESH Models API Reference

This section contains ESP-BLE-MESH Model related APIs, event types, event parameters, etc.

There are six categories of models:

- *Configuration Client/Server Models*
- *Health Client/Server Models*
- *Generic Client/Server Models*
- *Sensor Client/Server Models*
- *Time and Scenes Client/Server Models*
- *Lighting Client/Server Models*

Note: Definitions related to Server Models are being updated, and will be released soon.

Configuration Client/Server Models

Header File

- [bt/esp_ble_mesh/api/models/include/esp_ble_mesh_config_model_api.h](#)

Functions

`esp_err_t esp_ble_mesh_register_config_client_callback` (`esp_ble_mesh_cfg_client_cb_t callback`)

Register BLE Mesh Config Client Model callback.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] `callback`: Pointer to the callback function.

`esp_err_t esp_ble_mesh_register_config_server_callback` (`esp_ble_mesh_cfg_server_cb_t callback`)

Register BLE Mesh Config Server Model callback.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] `callback`: Pointer to the callback function.

esp_err_t **esp_ble_mesh_config_client_get_state** (*esp_ble_mesh_client_common_param_t* *params, *esp_ble_mesh_cfg_client_get_state_t* *get_state)

Get the value of Config Server Model states using the Config Client Model get messages.

Note If you want to find the opcodes and corresponding meanings accepted by this API, please refer to `esp_ble_mesh_opcode_config_client_get_t` in `esp_ble_mesh_defs.h`

Return ESP_OK on success or error code otherwise.

Parameters

- [in] `params`: Pointer to BLE Mesh common client parameters.
- [in] `get_state`: Pointer to a union, each kind of opcode corresponds to one structure inside. Shall not be set to NULL.

esp_err_t **esp_ble_mesh_config_client_set_state** (*esp_ble_mesh_client_common_param_t* *params, *esp_ble_mesh_cfg_client_set_state_t* *set_state)

Set the value of the Configuration Server Model states using the Config Client Model set messages.

Note If you want to find the opcodes and corresponding meanings accepted by this API, please refer to `esp_ble_mesh_opcode_config_client_set_t` in `esp_ble_mesh_defs.h`

Return ESP_OK on success or error code otherwise.

Parameters

- [in] `params`: Pointer to BLE Mesh common client parameters.
- [in] `set_state`: Pointer to a union, each kind of opcode corresponds to one structure inside. Shall not be set to NULL.

Unions

union `esp_ble_mesh_cfg_client_get_state_t`

```
#include <esp_ble_mesh_config_model_api.h> For ESP_BLE_MESH_MODEL_OP_BEACON_GET
ESP_BLE_MESH_MODEL_OP_COMPOSITION_DATA_GET ESP_BLE_MESH_MODEL_OP_DEFAULT_TTL_GET
ESP_BLE_MESH_MODEL_OP_GATT_PROXY_GET ESP_BLE_MESH_MODEL_OP_RELAY_GET
ESP_BLE_MESH_MODEL_OP_MODEL_PUB_GET ESP_BLE_MESH_MODEL_OP_FRIEND_GET
ESP_BLE_MESH_MODEL_OP_HEARTBEAT_PUB_GET ESP_BLE_MESH_MODEL_OP_HEARTBEAT_SUB_GET
the get_state parameter in the esp_ble_mesh_config_client_get_state function should not be set to NULL.
```

Public Members

esp_ble_mesh_cfg_model_pub_get_t **model_pub_get**

For ESP_BLE_MESH_MODEL_OP_MODEL_PUB_GET.

esp_ble_mesh_cfg_composition_data_get_t **comp_data_get**

For ESP_BLE_MESH_MODEL_OP_COMPOSITION_DATA_GET.

esp_ble_mesh_cfg_sig_model_sub_get_t **sig_model_sub_get**

For ESP_BLE_MESH_MODEL_OP_SIG_MODEL_SUB_GET

esp_ble_mesh_cfg_vnd_model_sub_get_t **vnd_model_sub_get**

For ESP_BLE_MESH_MODEL_OP_VENDOR_MODEL_SUB_GET

esp_ble_mesh_cfg_app_key_get_t **app_key_get**

For ESP_BLE_MESH_MODEL_OP_APP_KEY_GET.

esp_ble_mesh_cfg_node_identity_get_t **node_identity_get**

For ESP_BLE_MESH_MODEL_OP_NODE_IDENTITY_GET.

esp_ble_mesh_cfg_sig_model_app_get_t **sig_model_app_get**

For ESP_BLE_MESH_MODEL_OP_SIG_MODEL_APP_GET

esp_ble_mesh_cfg_vnd_model_app_get_t **vnd_model_app_get**

For ESP_BLE_MESH_MODEL_OP_VENDOR_MODEL_APP_GET

esp_ble_mesh_cfg_kr_phase_get_t **kr_phase_get**

For ESP_BLE_MESH_MODEL_OP_KEY_REFRESH_PHASE_GET

esp_ble_mesh_cfg_lpn_polltimeout_get_t **lpn_pollto_get**

For ESP_BLE_MESH_MODEL_OP_LPN_POLLTIMEOUT_GET

union esp_ble_mesh_cfg_client_set_state_t

```
#include <esp_ble_mesh_config_model_api.h> For ESP_BLE_MESH_MODEL_OP_BEACON_SET
ESP_BLE_MESH_MODEL_OP_DEFAULT_TTL_SET ESP_BLE_MESH_MODEL_OP_GATT_PROXY_SET
ESP_BLE_MESH_MODEL_OP_RELAY_SET ESP_BLE_MESH_MODEL_OP_MODEL_PUB_SET
ESP_BLE_MESH_MODEL_OP_MODEL_SUB_ADD ESP_BLE_MESH_MODEL_OP_MODEL_SUB_VIRTUAL_ADDR
ESP_BLE_MESH_MODEL_OP_MODEL_SUB_DELETE ESP_BLE_MESH_MODEL_OP_MODEL_SUB_VIRTUAL_ADDR
ESP_BLE_MESH_MODEL_OP_MODEL_SUB_OVERWRITE ESP_BLE_MESH_MODEL_OP_MODEL_SUB_VIRTUAL_ADDR
ESP_BLE_MESH_MODEL_OP_NET_KEY_ADD ESP_BLE_MESH_MODEL_OP_APP_KEY_ADD
ESP_BLE_MESH_MODEL_OP_MODEL_APP_BIND ESP_BLE_MESH_MODEL_OP_NODE_RESET
ESP_BLE_MESH_MODEL_OP_FRIEND_SET ESP_BLE_MESH_MODEL_OP_HEARTBEAT_PUB_SET
ESP_BLE_MESH_MODEL_OP_HEARTBEAT_SUB_SET the set_state parameter in the
esp_ble_mesh_config_client_set_state function should not be set to NULL.
```

Public Members

esp_ble_mesh_cfg_beacon_set_t **beacon_set**

For ESP_BLE_MESH_MODEL_OP_BEACON_SET

esp_ble_mesh_cfg_default_ttl_set_t **default_ttl_set**

For ESP_BLE_MESH_MODEL_OP_DEFAULT_TTL_SET

esp_ble_mesh_cfg_friend_set_t **friend_set**

For ESP_BLE_MESH_MODEL_OP_FRIEND_SET

esp_ble_mesh_cfg_gatt_proxy_set_t **gatt_proxy_set**

For ESP_BLE_MESH_MODEL_OP_GATT_PROXY_SET

esp_ble_mesh_cfg_relay_set_t **relay_set**

For ESP_BLE_MESH_MODEL_OP_RELAY_SET

esp_ble_mesh_cfg_net_key_add_t **net_key_add**

For ESP_BLE_MESH_MODEL_OP_NET_KEY_ADD

esp_ble_mesh_cfg_app_key_add_t **app_key_add**

For ESP_BLE_MESH_MODEL_OP_APP_KEY_ADD

esp_ble_mesh_cfg_model_app_bind_t **model_app_bind**

For ESP_BLE_MESH_MODEL_OP_MODEL_APP_BIND

esp_ble_mesh_cfg_model_pub_set_t **model_pub_set**

For ESP_BLE_MESH_MODEL_OP_MODEL_PUB_SET

esp_ble_mesh_cfg_model_sub_add_t **model_sub_add**

For ESP_BLE_MESH_MODEL_OP_MODEL_SUB_ADD

esp_ble_mesh_cfg_model_sub_delete_t **model_sub_delete**

For ESP_BLE_MESH_MODEL_OP_MODEL_SUB_DELETE

esp_ble_mesh_cfg_model_sub_overwrite_t **model_sub_overwrite**

For ESP_BLE_MESH_MODEL_OP_MODEL_SUB_OVERWRITE

esp_ble_mesh_cfg_model_sub_va_add_t **model_sub_va_add**

For ESP_BLE_MESH_MODEL_OP_MODEL_SUB_VIRTUAL_ADDR_ADD

esp_ble_mesh_cfg_model_sub_va_delete_t **model_sub_va_delete**

For ESP_BLE_MESH_MODEL_OP_MODEL_SUB_VIRTUAL_ADDR_DELETE

esp_ble_mesh_cfg_model_sub_va_overwrite_t **model_sub_va_overwrite**

For ESP_BLE_MESH_MODEL_OP_MODEL_SUB_VIRTUAL_ADDR_OVERWRITE

esp_ble_mesh_cfg_heartbeat_pub_set_t **heartbeat_pub_set**

For ESP_BLE_MESH_MODEL_OP_HEARTBEAT_PUB_SET

esp_ble_mesh_cfg_heartbeat_sub_set_t **heartbeat_sub_set**
For ESP_BLE_MESH_MODEL_OP_HEARTBEAT_SUB_SET

esp_ble_mesh_cfg_model_pub_va_set_t **model_pub_va_set**
For ESP_BLE_MESH_MODEL_OP_MODEL_PUB_VIRTUAL_ADDR_SET

esp_ble_mesh_cfg_model_sub_delete_all_t **model_sub_delete_all**
For ESP_BLE_MESH_MODEL_OP_MODEL_SUB_DELETE_ALL

esp_ble_mesh_cfg_net_key_update_t **net_key_update**
For ESP_BLE_MESH_MODEL_OP_NET_KEY_UPDATE

esp_ble_mesh_cfg_net_key_delete_t **net_key_delete**
For ESP_BLE_MESH_MODEL_OP_NET_KEY_DELETE

esp_ble_mesh_cfg_app_key_update_t **app_key_update**
For ESP_BLE_MESH_MODEL_OP_APP_KEY_UPDATE

esp_ble_mesh_cfg_app_key_delete_t **app_key_delete**
For ESP_BLE_MESH_MODEL_OP_APP_KEY_DELETE

esp_ble_mesh_cfg_node_identity_set_t **node_identity_set**
For ESP_BLE_MESH_MODEL_OP_NODE_IDENTITY_SET

esp_ble_mesh_cfg_model_app_unbind_t **model_app_unbind**
For ESP_BLE_MESH_MODEL_OP_MODEL_APP_UNBIND

esp_ble_mesh_cfg_kr_phase_set_t **kr_phase_set**
For ESP_BLE_MESH_MODEL_OP_KEY_REFRESH_PHASE_SET

esp_ble_mesh_cfg_net_transmit_set_t **net_transmit_set**
For ESP_BLE_MESH_MODEL_OP_NETWORK_TRANSMIT_SET

union esp_ble_mesh_cfg_client_common_cb_param_t
#include <esp_ble_mesh_config_model_api.h> Configuration Client Model received message union.

Public Members

esp_ble_mesh_cfg_beacon_status_cb_t **beacon_status**
The beacon status value

esp_ble_mesh_cfg_comp_data_status_cb_t **comp_data_status**
The composition data status value

esp_ble_mesh_cfg_default_ttl_status_cb_t **default_ttl_status**
The default_ttl status value

esp_ble_mesh_cfg_gatt_proxy_status_cb_t **gatt_proxy_status**
The gatt_proxy status value

esp_ble_mesh_cfg_relay_status_cb_t **relay_status**
The relay status value

esp_ble_mesh_cfg_model_pub_status_cb_t **model_pub_status**
The model publication status value

esp_ble_mesh_cfg_model_sub_status_cb_t **model_sub_status**
The model subscription status value

esp_ble_mesh_cfg_net_key_status_cb_t **netkey_status**
The netkey status value

esp_ble_mesh_cfg_app_key_status_cb_t **appkey_status**
The appkey status value

esp_ble_mesh_cfg_mod_app_status_cb_t **model_app_status**
The model app status value

esp_ble_mesh_cfg_friend_status_cb_t **friend_status**

The friend status value

esp_ble_mesh_cfg_hb_pub_status_cb_t **heartbeat_pub_status**

The heartbeat publication status value

esp_ble_mesh_cfg_hb_sub_status_cb_t **heartbeat_sub_status**

The heartbeat subscription status value

esp_ble_mesh_cfg_net_trans_status_cb_t **net_transmit_status**

The network transmit status value

esp_ble_mesh_cfg_model_sub_list_cb_t **model_sub_list**

The model subscription list value

esp_ble_mesh_cfg_net_key_list_cb_t **netkey_list**

The network key index list value

esp_ble_mesh_cfg_app_key_list_cb_t **appkey_list**

The application key index list value

esp_ble_mesh_cfg_node_id_status_cb_t **node_identity_status**

The node identity status value

esp_ble_mesh_cfg_model_app_list_cb_t **model_app_list**

The model application key index list value

esp_ble_mesh_cfg_kr_phase_status_cb_t **kr_phase_status**

The key refresh phase status value

esp_ble_mesh_cfg_lpn_pollto_status_cb_t **lpn_timeout_status**

The low power node poll timeout status value

union esp_ble_mesh_cfg_server_state_change_t

#include <esp_ble_mesh_config_model_api.h> Configuration Server model state change value union.

Public Members

esp_ble_mesh_state_change_cfg_mod_pub_set_t **mod_pub_set**

The *recv_op* in *ctx* can be used to decide which state is changed. Config Model Publication Set

esp_ble_mesh_state_change_cfg_model_sub_add_t **mod_sub_add**

Config Model Subscription Add

esp_ble_mesh_state_change_cfg_model_sub_delete_t **mod_sub_delete**

Config Model Subscription Delete

esp_ble_mesh_state_change_cfg_netkey_add_t **netkey_add**

Config NetKey Add

esp_ble_mesh_state_change_cfg_netkey_update_t **netkey_update**

Config NetKey Update

esp_ble_mesh_state_change_cfg_netkey_delete_t **netkey_delete**

Config NetKey Delete

esp_ble_mesh_state_change_cfg_appkey_add_t **appkey_add**

Config AppKey Add

esp_ble_mesh_state_change_cfg_appkey_update_t **appkey_update**

Config AppKey Update

esp_ble_mesh_state_change_cfg_appkey_delete_t **appkey_delete**

Config AppKey Delete

esp_ble_mesh_state_change_cfg_model_app_bind_t **mod_app_bind**

Config Model App Bind

esp_ble_mesh_state_change_cfg_model_app_unbind_t **mod_app_unbind**
Config Model App Unbind

esp_ble_mesh_state_change_cfg_kr_phase_set_t **kr_phase_set**
Config Key Refresh Phase Set

union esp_ble_mesh_cfg_server_cb_value_t
#include <esp_ble_mesh_config_model_api.h> Configuration Server model callback value union.

Public Members

esp_ble_mesh_cfg_server_state_change_t **state_change**
ESP_BLE_MESH_CFG_SERVER_STATE_CHANGE_EVT

Structures

struct esp_ble_mesh_cfg_srv
Configuration Server Model context

Public Members

esp_ble_mesh_model_t ***model**
Pointer to Configuration Server Model

uint8_t **net_transmit**
Network Transmit state

uint8_t **relay**
Relay Mode state

uint8_t **relay_retransmit**
Relay Retransmit state

uint8_t **beacon**
Secure Network Beacon state

uint8_t **gatt_proxy**
GATT Proxy state

uint8_t **friend_state**
Friend state

uint8_t **default_ttl**
Default TTL

struct k_delayed_work timer
Heartbeat Publication timer

uint16_t **dst**
Destination address for Heartbeat messages

uint16_t **count**
Number of Heartbeat messages to be sent
Number of Heartbeat messages received

uint8_t **period**
Period for sending Heartbeat messages

uint8_t **ttl**
TTL to be used when sending Heartbeat messages

uint16_t **feature**
Bit field indicating features that trigger Heartbeat messages when changed

uint16_t net_idx
NetKey Index used by Heartbeat Publication

struct esp_ble_mesh_cfg_srv::[anonymous] heartbeat_pub
Heartbeat Publication

int64_t expiry
Timestamp when Heartbeat subscription period is expired

uint16_t src
Source address for Heartbeat messages

uint8_t min_hops
Minimum hops when receiving Heartbeat messages

uint8_t max_hops
Maximum hops when receiving Heartbeat messages

esp_ble_mesh_cb_t heartbeat_recv_cb
Optional heartbeat subscription tracking function

struct esp_ble_mesh_cfg_srv::[anonymous] heartbeat_sub
Heartbeat Subscription

struct esp_ble_mesh_cfg_composition_data_get_t
Parameters of Config Composition Data Get.

Public Members

uint8_t page
Page number of the Composition Data.

struct esp_ble_mesh_cfg_model_pub_get_t
Parameters of Config Model Publication Get.

Public Members

uint16_t element_addr
The element address

uint16_t model_id
The model id

uint16_t company_id
The company id, if not a vendor model, shall set to 0xFFFF

struct esp_ble_mesh_cfg_sig_model_sub_get_t
Parameters of Config SIG Model Subscription Get.

Public Members

uint16_t element_addr
The element address

uint16_t model_id
The model id

struct esp_ble_mesh_cfg_vnd_model_sub_get_t
Parameters of Config Vendor Model Subscription Get.

Public Members**uint16_t element_addr**

The element address

uint16_t model_id

The model id

uint16_t company_id

The company id, if not a vendor model, shall set to 0xFFFF

struct esp_ble_mesh_cfg_app_key_get_t

Parameters of Config AppKey Get.

Public Members**uint16_t net_idx**

The network key index

struct esp_ble_mesh_cfg_node_identity_get_t

Parameters of Config Node Identity Get.

Public Members**uint16_t net_idx**

The network key index

struct esp_ble_mesh_cfg_sig_model_app_get_t

Parameters of Config SIG Model App Get.

Public Members**uint16_t element_addr**

The element address

uint16_t model_id

The model id

struct esp_ble_mesh_cfg_vnd_model_app_get_t

Parameters of Config Vendor Model App Get.

Public Members**uint16_t element_addr**

The element address

uint16_t model_id

The model id

uint16_t company_id

The company id, if not a vendor model, shall set to 0xFFFF

struct esp_ble_mesh_cfg_kr_phase_get_t

Parameters of Config Key Refresh Phase Get.

Public Members**uint16_t net_idx**

The network key index

struct esp_ble_mesh_cfg_lpn_polltimeout_get_t

Parameters of Config Low Power Node PollTimeout Get.

Public Members**uint16_t lpn_addr**

The unicast address of the Low Power node

struct esp_ble_mesh_cfg_beacon_set_t

Parameters of Config Beacon Set.

Public Members**uint8_t beacon**

New Secure Network Beacon state

struct esp_ble_mesh_cfg_default_ttl_set_t

Parameters of Config Default TTL Set.

Public Members**uint8_t ttl**

The default TTL state value

struct esp_ble_mesh_cfg_friend_set_t

Parameters of Config Friend Set.

Public Members**uint8_t friend_state**

The friend state value

struct esp_ble_mesh_cfg_gatt_proxy_set_t

Parameters of Config GATT Proxy Set.

Public Members**uint8_t gatt_proxy**

The GATT Proxy state value

struct esp_ble_mesh_cfg_relay_set_t

Parameters of Config Relay Set.

Public Members**uint8_t relay**

The relay value

uint8_t relay_retransmit

The relay retransmit value

struct esp_ble_mesh_cfg_net_key_add_t

Parameters of Config NetKey Add.

Public Members

`uint16_t net_idx`
The network key index

`uint8_t net_key[16]`
The network key value

struct esp_ble_mesh_cfg_app_key_add_t
Parameters of Config AppKey Add.

Public Members

`uint16_t net_idx`
The network key index

`uint16_t app_idx`
The app key index

`uint8_t app_key[16]`
The app key value

struct esp_ble_mesh_cfg_model_app_bind_t
Parameters of Config Model App Bind.

Public Members

`uint16_t element_addr`
The element address

`uint16_t model_app_idx`
Index of the app key to bind with the model

`uint16_t model_id`
The model id

`uint16_t company_id`
The company id, if not a vendor model, shall set to 0xFFFF

struct esp_ble_mesh_cfg_model_pub_set_t
Parameters of Config Model Publication Set.

Public Members

`uint16_t element_addr`
The element address

`uint16_t publish_addr`
Value of the publish address

`uint16_t publish_app_idx`
Index of the application key

`bool cred_flag`
Value of the Friendship Credential Flag

`uint8_t publish_ttl`
Default TTL value for the publishing messages

`uint8_t publish_period`
Period for periodic status publishing

`uint8_t publish_retransmit`

Number of retransmissions and number of 50-millisecond steps between retransmissions

`uint16_t model_id`

The model id

`uint16_t company_id`

The company id, if not a vendor model, shall set to 0xFFFF

struct esp_ble_mesh_cfg_model_sub_add_t

Parameters of Config Model Subscription Add.

Public Members

`uint16_t element_addr`

The element address

`uint16_t sub_addr`

The address to be added to the Subscription List

`uint16_t model_id`

The model id

`uint16_t company_id`

The company id, if not a vendor model, shall set to 0xFFFF

struct esp_ble_mesh_cfg_model_sub_delete_t

Parameters of Config Model Subscription Delete.

Public Members

`uint16_t element_addr`

The element address

`uint16_t sub_addr`

The address to be removed from the Subscription List

`uint16_t model_id`

The model id

`uint16_t company_id`

The company id, if not a vendor model, shall set to 0xFFFF

struct esp_ble_mesh_cfg_model_sub_overwrite_t

Parameters of Config Model Subscription Overwrite.

Public Members

`uint16_t element_addr`

The element address

`uint16_t sub_addr`

The address to be added to the Subscription List

`uint16_t model_id`

The model id

`uint16_t company_id`

The company id, if not a vendor model, shall set to 0xFFFF

struct esp_ble_mesh_cfg_model_sub_va_add_t

Parameters of Config Model Subscription Virtual Address Add.

Public Members**uint16_t element_addr**

The element address

uint8_t label_uuid[16]

The Label UUID of the virtual address to be added to the Subscription List

uint16_t model_id

The model id

uint16_t company_id

The company id, if not a vendor model, shall set to 0xFFFF

struct esp_ble_mesh_cfg_model_sub_va_delete_t

Parameters of Config Model Subscription Virtual Address Delete.

Public Members**uint16_t element_addr**

The element address

uint8_t label_uuid[16]

The Label UUID of the virtual address to be removed from the Subscription List

uint16_t model_id

The model id

uint16_t company_id

The company id, if not a vendor model, shall set to 0xFFFF

struct esp_ble_mesh_cfg_model_sub_va_overwrite_t

Parameters of Config Model Subscription Virtual Address Overwrite.

Public Members**uint16_t element_addr**

The element address

uint8_t label_uuid[16]

The Label UUID of the virtual address to be added to the Subscription List

uint16_t model_id

The model id

uint16_t company_id

The company id, if not a vendor model, shall set to 0xFFFF

struct esp_ble_mesh_cfg_model_pub_va_set_t

Parameters of Config Model Publication Virtual Address Set.

Public Members**uint16_t element_addr**

The element address

uint8_t label_uuid[16]

Value of the Label UUID publish address

uint16_t publish_app_idx

Index of the application key

bool **cred_flag**
Value of the Friendship Credential Flag

uint8_t **publish_ttl**
Default TTL value for the publishing messages

uint8_t **publish_period**
Period for periodic status publishing

uint8_t **publish_retransmit**
Number of retransmissions and number of 50-millisecond steps between retransmissions

uint16_t **model_id**
The model id

uint16_t **company_id**
The company id, if not a vendor model, shall set to 0xFFFF

struct esp_ble_mesh_cfg_model_sub_delete_all_t
Parameters of Config Model Subscription Delete All.

Public Members

uint16_t **element_addr**
The element address

uint16_t **model_id**
The model id

uint16_t **company_id**
The company id, if not a vendor model, shall set to 0xFFFF

struct esp_ble_mesh_cfg_net_key_update_t
Parameters of Config NetKey Update.

Public Members

uint16_t **net_idx**
The network key index

uint8_t **net_key**[16]
The network key value

struct esp_ble_mesh_cfg_net_key_delete_t
Parameters of Config NetKey Delete.

Public Members

uint16_t **net_idx**
The network key index

struct esp_ble_mesh_cfg_app_key_update_t
Parameters of Config AppKey Update.

Public Members

uint16_t **net_idx**
The network key index

uint16_t **app_idx**
The app key index

`uint8_t app_key[16]`
The app key value

struct esp_ble_mesh_cfg_app_key_delete_t
Parameters of Config AppKey Delete.

Public Members

`uint16_t net_idx`
The network key index

`uint16_t app_idx`
The app key index

struct esp_ble_mesh_cfg_node_identity_set_t
Parameters of Config Node Identity Set.

Public Members

`uint16_t net_idx`
The network key index

`uint8_t identity`
New Node Identity state

struct esp_ble_mesh_cfg_model_app_unbind_t
Parameters of Config Model App Unbind.

Public Members

`uint16_t element_addr`
The element address

`uint16_t model_app_idx`
Index of the app key to bind with the model

`uint16_t model_id`
The model id

`uint16_t company_id`
The company id, if not a vendor model, shall set to 0xFFFF

struct esp_ble_mesh_cfg_kr_phase_set_t
Parameters of Config Key Refresh Phase Set.

Public Members

`uint16_t net_idx`
The network key index

`uint8_t transition`
New Key Refresh Phase Transition

struct esp_ble_mesh_cfg_net_transmit_set_t
Parameters of Config Network Transmit Set.

Public Members

uint8_t net_transmit
Network Transmit State

struct esp_ble_mesh_cfg_heartbeat_pub_set_t
Parameters of Config Model Heartbeat Publication Set.

Public Members

uint16_t dst
Destination address for Heartbeat messages

uint8_t count
Number of Heartbeat messages to be sent

uint8_t period
Period for sending Heartbeat messages

uint8_t ttl
TTL to be used when sending Heartbeat messages

uint16_t feature
Bit field indicating features that trigger Heartbeat messages when changed

uint16_t net_idx
NetKey Index

struct esp_ble_mesh_cfg_heartbeat_sub_set_t
Parameters of Config Model Heartbeat Subscription Set.

Public Members

uint16_t src
Source address for Heartbeat messages

uint16_t dst
Destination address for Heartbeat messages

uint8_t period
Period for receiving Heartbeat messages

struct esp_ble_mesh_cfg_beacon_status_cb_t
Parameter of Config Beacon Status

Public Members

uint8_t beacon
Secure Network Beacon state value

struct esp_ble_mesh_cfg_comp_data_status_cb_t
Parameters of Config Composition Data Status

Public Members

uint8_t page
Page number of the Composition Data

struct net_buf_simple *composition_data
Pointer to Composition Data for the identified page

struct esp_ble_mesh_cfg_default_ttl_status_cb_t
Parameter of Config Default TTL Status

Public Members

uint8_t default_ttl
Default TTL state value

struct esp_ble_mesh_cfg_gatt_proxy_status_cb_t
Parameter of Config GATT Proxy Status

Public Members

uint8_t gatt_proxy
GATT Proxy state value

struct esp_ble_mesh_cfg_relay_status_cb_t
Parameters of Config Relay Status

Public Members

uint8_t relay
Relay state value

uint8_t retransmit
Relay retransmit value(number of retransmissions and number of 10-millisecond steps between retransmissions)

struct esp_ble_mesh_cfg_model_pub_status_cb_t
Parameters of Config Model Publication Status

Public Members

uint8_t status
Status Code for the request message

uint16_t element_addr
Address of the element

uint16_t publish_addr
Value of the publish address

uint16_t app_idx
Index of the application key

bool cred_flag
Value of the Friendship Credential Flag

uint8_t ttl
Default TTL value for the outgoing messages

uint8_t period
Period for periodic status publishing

uint8_t transmit
Number of retransmissions and number of 50-millisecond steps between retransmissions

uint16_t company_id
Company ID

uint16_t model_id
Model ID

struct esp_ble_mesh_cfg_model_sub_status_cb_t
Parameters of Config Model Subscription Status

Public Members

uint8_t status
Status Code for the request message

uint16_t element_addr
Address of the element

uint16_t sub_addr
Value of the address

uint16_t company_id
Company ID

uint16_t model_id
Model ID

struct esp_ble_mesh_cfg_net_key_status_cb_t
Parameters of Config NetKey Status

Public Members

uint8_t status
Status Code for the request message

uint16_t net_idx
Index of the NetKey

struct esp_ble_mesh_cfg_app_key_status_cb_t
Parameters of Config AppKey Status

Public Members

uint8_t status
Status Code for the request message

uint16_t net_idx
Index of the NetKey

uint16_t app_idx
Index of the application key

struct esp_ble_mesh_cfg_mod_app_status_cb_t
Parameters of Config Model App Status

Public Members

uint8_t status
Status Code for the request message

uint16_t element_addr
Address of the element

uint16_t app_idx
Index of the application key

uint16_t company_id
Company ID

`uint16_t model_id`
Model ID

`struct esp_ble_mesh_cfg_friend_status_cb_t`
Parameter of Config Friend Status

Public Members

`uint8_t friend_state`
Friend state value

`struct esp_ble_mesh_cfg_hb_pub_status_cb_t`
Parameters of Config Heartbeat Publication Status

Public Members

`uint8_t status`
Status Code for the request message

`uint16_t dst`
Destination address for Heartbeat messages

`uint8_t count`
Number of Heartbeat messages remaining to be sent

`uint8_t period`
Period for sending Heartbeat messages

`uint8_t ttl`
TTL to be used when sending Heartbeat messages

`uint16_t features`
Features that trigger Heartbeat messages when changed

`uint16_t net_idx`
Index of the NetKey

`struct esp_ble_mesh_cfg_hb_sub_status_cb_t`
Parameters of Config Heartbeat Subscription Status

Public Members

`uint8_t status`
Status Code for the request message

`uint16_t src`
Source address for Heartbeat messages

`uint16_t dst`
Destination address for Heartbeat messages

`uint8_t period`
Remaining Period for processing Heartbeat messages

`uint8_t count`
Number of Heartbeat messages received

`uint8_t min_hops`
Minimum hops when receiving Heartbeat messages

`uint8_t max_hops`
Maximum hops when receiving Heartbeat messages

struct esp_ble_mesh_cfg_net_trans_status_cb_t
Parameters of Config Network Transmit Status

Public Members

uint8_t net_trans_count : 3
Number of transmissions for each Network PDU originating from the node

uint8_t net_trans_step : 5
Maximum hops when receiving Heartbeat messages

struct esp_ble_mesh_cfg_model_sub_list_cb_t
Parameters of Config SIG/Vendor Subscription List

Public Members

uint8_t status
Status Code for the request message

uint16_t element_addr
Address of the element

uint16_t company_id
Company ID

uint16_t model_id
Model ID

struct net_buf_simple *sub_addr
A block of all addresses from the Subscription List

struct esp_ble_mesh_cfg_net_key_list_cb_t
Parameter of Config NetKey List

Public Members

struct net_buf_simple *net_idx
A list of NetKey Indexes known to the node

struct esp_ble_mesh_cfg_app_key_list_cb_t
Parameters of Config AppKey List

Public Members

uint8_t status
Status Code for the request message

uint16_t net_idx
NetKey Index of the NetKey that the AppKeys are bound to

struct net_buf_simple *app_idx
A list of AppKey indexes that are bound to the NetKey identified by NetKeyIndex

struct esp_ble_mesh_cfg_node_id_status_cb_t
Parameters of Config Node Identity Status

Public Members

uint8_t status
Status Code for the request message

uint16_t net_idx
Index of the NetKey

uint8_t identity
Node Identity state

struct esp_ble_mesh_cfg_model_app_list_cb_t
Parameters of Config SIG/Vendor Model App List

Public Members

uint8_t status
Status Code for the request message

uint16_t element_addr
Address of the element

uint16_t company_id
Company ID

uint16_t model_id
Model ID

struct net_buf_simple *app_idx
All AppKey indexes bound to the Model

struct esp_ble_mesh_cfg_kr_phase_status_cb_t
Parameters of Config Key Refresh Phase Status

Public Members

uint8_t status
Status Code for the request message

uint16_t net_idx
Index of the NetKey

uint8_t phase
Key Refresh Phase state

struct esp_ble_mesh_cfg_lpn_pollto_status_cb_t
Parameters of Config Low Power Node PollTimeout Status

Public Members

uint16_t lpn_addr
The unicast address of the Low Power node

int32_t poll_timeout
The current value of the PollTimeout timer of the Low Power node

struct esp_ble_mesh_cfg_client_cb_param_t
Configuration Client Model callback parameters

Public Members**int error_code**

Appropriate error code

***esp_ble_mesh_client_common_param_t* *params**

The client common parameters

***esp_ble_mesh_cfg_client_common_cb_param_t* status_cb**

The config status message callback values

struct esp_ble_mesh_state_change_cfg_mod_pub_set_t

Configuration Server model related context.

Public Members**uint16_t element_addr**

Element Address

uint16_t pub_addr

Publish Address

uint16_t app_idx

AppKey Index

bool cred_flag

Friendship Credential Flag

uint8_t pub_ttl

Publish TTL

uint8_t pub_period

Publish Period

uint8_t pub_retransmit

Publish Retransmit

uint16_t company_id

Company ID

uint16_t model_id

Model ID

struct esp_ble_mesh_state_change_cfg_model_sub_add_t

Parameters of Config Model Subscription Add

Public Members**uint16_t element_addr**

Element Address

uint16_t sub_addr

Subscription Address

uint16_t company_id

Company ID

uint16_t model_id

Model ID

struct esp_ble_mesh_state_change_cfg_model_sub_delete_t

Parameters of Config Model Subscription Delete

Public Members

`uint16_t element_addr`
Element Address

`uint16_t sub_addr`
Subscription Address

`uint16_t company_id`
Company ID

`uint16_t model_id`
Model ID

`struct esp_ble_mesh_state_change_cfg_netkey_add_t`
Parameters of Config NetKey Add

Public Members

`uint16_t net_idx`
NetKey Index

`uint8_t net_key[16]`
NetKey

`struct esp_ble_mesh_state_change_cfg_netkey_update_t`
Parameters of Config NetKey Update

Public Members

`uint16_t net_idx`
NetKey Index

`uint8_t net_key[16]`
NetKey

`struct esp_ble_mesh_state_change_cfg_netkey_delete_t`
Parameter of Config NetKey Delete

Public Members

`uint16_t net_idx`
NetKey Index

`struct esp_ble_mesh_state_change_cfg_appkey_add_t`
Parameters of Config AppKey Add

Public Members

`uint16_t net_idx`
NetKey Index

`uint16_t app_idx`
AppKey Index

`uint8_t app_key[16]`
AppKey

`struct esp_ble_mesh_state_change_cfg_appkey_update_t`
Parameters of Config AppKey Update

Public Members

`uint16_t net_idx`
NetKey Index

`uint16_t app_idx`
AppKey Index

`uint8_t app_key[16]`
AppKey

struct esp_ble_mesh_state_change_cfg_appkey_delete_t
Parameters of Config AppKey Delete

Public Members

`uint16_t net_idx`
NetKey Index

`uint16_t app_idx`
AppKey Index

struct esp_ble_mesh_state_change_cfg_model_app_bind_t
Parameters of Config Model App Bind

Public Members

`uint16_t element_addr`
Element Address

`uint16_t app_idx`
AppKey Index

`uint16_t company_id`
Company ID

`uint16_t model_id`
Model ID

struct esp_ble_mesh_state_change_cfg_model_app_unbind_t
Parameters of Config Model App Unbind

Public Members

`uint16_t element_addr`
Element Address

`uint16_t app_idx`
AppKey Index

`uint16_t company_id`
Company ID

`uint16_t model_id`
Model ID

struct esp_ble_mesh_state_change_cfg_kr_phase_set_t
Parameters of Config Key Refresh Phase Set

Public Members

`uint16_t net_idx`

NetKey Index

`uint8_t kr_phase`

New Key Refresh Phase Transition

struct `esp_ble_mesh_cfg_server_cb_param_t`

Configuration Server model callback parameters

Public Members

`esp_ble_mesh_model_t *model`

Pointer to the server model structure

`esp_ble_mesh_msg_ctx_t ctx`

Context of the received message

`esp_ble_mesh_cfg_server_cb_value_t value`

Value of the received configuration messages

Macros

ESP_BLE_MESH_MODEL_CFG_SRV (`srv_data`)

Define a new Config Server Model.

Note The Config Server Model can only be included by a Primary Element.

Return New Config Server Model instance.

Parameters

- `srv_data`: Pointer to a unique Config Server Model `user_data`.

ESP_BLE_MESH_MODEL_CFG_CLI (`cli_data`)

Define a new Config Client Model.

Note The Config Client Model can only be included by a Primary Element.

Return New Config Client Model instance.

Parameters

- `cli_data`: Pointer to a unique struct `esp_ble_mesh_client_t`.

Type Definitions

typedef struct `esp_ble_mesh_cfg_srv` `esp_ble_mesh_cfg_srv_t`

Configuration Server Model context

typedef void (`*esp_ble_mesh_cfg_client_cb_t`) (`esp_ble_mesh_cfg_client_cb_event_t` `event`,
`esp_ble_mesh_cfg_client_cb_param_t`
`*param`)

Bluetooth Mesh Config Client and Server Model functions.

Configuration Client Model callback function type

Parameters

- `event`: Event type
- `param`: Pointer to callback parameter

typedef void (`*esp_ble_mesh_cfg_server_cb_t`) (`esp_ble_mesh_cfg_server_cb_event_t` `event`,
`esp_ble_mesh_cfg_server_cb_param_t`
`*param`)

Configuration Server Model callback function type.

Parameters

- `event`: Event type
- `param`: Pointer to callback parameter

Enumerations

enum esp_ble_mesh_cfg_client_cb_event_t

This enum value is the event of Configuration Client Model

Values:

ESP_BLE_MESH_CFG_CLIENT_GET_STATE_EVT

ESP_BLE_MESH_CFG_CLIENT_SET_STATE_EVT

ESP_BLE_MESH_CFG_CLIENT_PUBLISH_EVT

ESP_BLE_MESH_CFG_CLIENT_TIMEOUT_EVT

ESP_BLE_MESH_CFG_CLIENT_EVT_MAX

enum esp_ble_mesh_cfg_server_cb_event_t

This enum value is the event of Configuration Server model

Values:

ESP_BLE_MESH_CFG_SERVER_STATE_CHANGE_EVT

ESP_BLE_MESH_CFG_SERVER_EVT_MAX

Health Client/Server Models

Header File

- [bt/esp_ble_mesh/api/models/include/esp_ble_mesh_generic_model_api.h](#)

Functions

esp_err_t esp_ble_mesh_register_generic_client_callback (*esp_ble_mesh_generic_client_cb_t* callback)

Register BLE Mesh Generic Client Model callback.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] callback: Pointer to the callback function.

esp_err_t esp_ble_mesh_generic_client_get_state (*esp_ble_mesh_client_common_param_t* *params, *esp_ble_mesh_generic_client_get_state_t* *get_state)

Get the value of Generic Server Model states using the Generic Client Model get messages.

Note If you want to find the opcodes and corresponding meanings accepted by this API, please refer to `esp_ble_mesh_generic_message_opcode_t` in `esp_ble_mesh_defs.h`

Return ESP_OK on success or error code otherwise.

Parameters

- [in] params: Pointer to BLE Mesh common client parameters.
- [in] get_state: Pointer to generic get message value. Shall not be set to NULL.

esp_err_t esp_ble_mesh_generic_client_set_state (*esp_ble_mesh_client_common_param_t* *params, *esp_ble_mesh_generic_client_set_state_t* *set_state)

Set the value of Generic Server Model states using the Generic Client Model set messages.

Note If you want to find the opcodes and corresponding meanings accepted by this API, please refer to `esp_ble_mesh_generic_message_opcode_t` in `esp_ble_mesh_defs.h`

Return ESP_OK on success or error code otherwise.

Parameters

- [in] params: Pointer to BLE Mesh common client parameters.
- [in] set_state: Pointer to generic set message value. Shall not be set to NULL.

esp_err_t **esp_ble_mesh_register_generic_server_callback** (*esp_ble_mesh_generic_server_cb_t* callback)

Register BLE Mesh Generic Server Model callback.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] callback: Pointer to the callback function.

Unions

union esp_ble_mesh_generic_client_get_state_t

#include <esp_ble_mesh_generic_model_api.h> Generic Client Model get message union.

Public Members

esp_ble_mesh_gen_user_property_get_t **user_property_get**

For ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTY_GET

esp_ble_mesh_gen_admin_property_get_t **admin_property_get**

For ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTY_GET

esp_ble_mesh_gen_manufacturer_property_get_t **manufacturer_property_get**

For ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTY_SET

esp_ble_mesh_gen_client_properties_get_t **client_properties_get**

For ESP_BLE_MESH_MODEL_OP_GEN_CLIENT_PROPERTIES_GET

union esp_ble_mesh_generic_client_set_state_t

#include <esp_ble_mesh_generic_model_api.h> Generic Client Model set message union.

Public Members

esp_ble_mesh_gen_onoff_set_t **onoff_set**

For ESP_BLE_MESH_MODEL_OP_GEN_ONOFF_SET & ESP_BLE_MESH_MODEL_OP_GEN_ONOFF_SET_UNACK

esp_ble_mesh_gen_level_set_t **level_set**

For ESP_BLE_MESH_MODEL_OP_GEN_LEVEL_SET & ESP_BLE_MESH_MODEL_OP_GEN_LEVEL_SET_UNACK

esp_ble_mesh_gen_delta_set_t **delta_set**

For ESP_BLE_MESH_MODEL_OP_GEN_DELTA_SET & ESP_BLE_MESH_MODEL_OP_GEN_DELTA_SET_UNACK

esp_ble_mesh_gen_move_set_t **move_set**

For ESP_BLE_MESH_MODEL_OP_GEN_MOVE_SET & ESP_BLE_MESH_MODEL_OP_GEN_MOVE_SET_UNACK

esp_ble_mesh_gen_def_trans_time_set_t **def_trans_time_set**

For ESP_BLE_MESH_MODEL_OP_GEN_DEF_TRANS_TIME_SET &
ESP_BLE_MESH_MODEL_OP_GEN_DEF_TRANS_TIME_SET_UNACK

esp_ble_mesh_gen_onpowerup_set_t **power_set**

For ESP_BLE_MESH_MODEL_OP_GEN_ONPOWERUP_SET &
ESP_BLE_MESH_MODEL_OP_GEN_ONPOWERUP_SET_UNACK

esp_ble_mesh_gen_power_level_set_t **power_level_set**

For ESP_BLE_MESH_MODEL_OP_GEN_POWER_LEVEL_SET &
ESP_BLE_MESH_MODEL_OP_GEN_POWER_LEVEL_SET_UNACK

esp_ble_mesh_gen_power_default_set_t **power_default_set**

For ESP_BLE_MESH_MODEL_OP_GEN_POWER_DEFAULT_SET &
ESP_BLE_MESH_MODEL_OP_GEN_POWER_DEFAULT_SET_UNACK

esp_ble_mesh_gen_power_range_set_t **power_range_set**

For ESP_BLE_MESH_MODEL_OP_GEN_POWER_RANGE_SET &
ESP_BLE_MESH_MODEL_OP_GEN_POWER_RANGE_SET_UNACK

```

esp_ble_mesh_gen_loc_global_set_t loc_global_set
    For ESP_BLE_MESH_MODEL_OP_GEN_LOC_GLOBAL_SET &
    ESP_BLE_MESH_MODEL_OP_GEN_LOC_GLOBAL_SET_UNACK

esp_ble_mesh_gen_loc_local_set_t loc_local_set
    For ESP_BLE_MESH_MODEL_OP_GEN_LOC_LOCAL_SET &
    ESP_BLE_MESH_MODEL_OP_GEN_LOC_LOCAL_SET_UNACK

esp_ble_mesh_gen_user_property_set_t user_property_set
    For ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTY_SET &
    ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTY_SET_UNACK

esp_ble_mesh_gen_admin_property_set_t admin_property_set
    For ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTY_SET &
    ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTY_SET_UNACK

esp_ble_mesh_gen_manufacturer_property_set_t manufacturer_property_set
    For ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTY_SET &
    ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTY_SET_UNACK

```

```

union esp_ble_mesh_gen_client_status_cb_t
#include <esp_ble_mesh_generic_model_api.h> Generic Client Model received message union.

```

Public Members

```

esp_ble_mesh_gen_onoff_status_cb_t onoff_status
    For ESP_BLE_MESH_MODEL_OP_GEN_ONOFF_STATUS

esp_ble_mesh_gen_level_status_cb_t level_status
    For ESP_BLE_MESH_MODEL_OP_GEN_LEVEL_STATUS

esp_ble_mesh_gen_def_trans_time_status_cb_t def_trans_time_status
    For ESP_BLE_MESH_MODEL_OP_GEN_DEF_TRANS_TIME_STATUS

esp_ble_mesh_gen_onpowerup_status_cb_t onpowerup_status
    For ESP_BLE_MESH_MODEL_OP_GEN_ONPOWERUP_STATUS

esp_ble_mesh_gen_power_level_status_cb_t power_level_status
    For ESP_BLE_MESH_MODEL_OP_GEN_POWER_LEVEL_STATUS

esp_ble_mesh_gen_power_last_status_cb_t power_last_status
    For ESP_BLE_MESH_MODEL_OP_GEN_POWER_LAST_STATUS

esp_ble_mesh_gen_power_default_status_cb_t power_default_status
    For ESP_BLE_MESH_MODEL_OP_GEN_POWER_DEFAULT_STATUS

esp_ble_mesh_gen_power_range_status_cb_t power_range_status
    For ESP_BLE_MESH_MODEL_OP_GEN_POWER_RANGE_STATUS

esp_ble_mesh_gen_battery_status_cb_t battery_status
    For ESP_BLE_MESH_MODEL_OP_GEN_BATTERY_STATUS

esp_ble_mesh_gen_loc_global_status_cb_t location_global_status
    For ESP_BLE_MESH_MODEL_OP_GEN_LOC_GLOBAL_STATUS

esp_ble_mesh_gen_loc_local_status_cb_t location_local_status
    ESP_BLE_MESH_MODEL_OP_GEN_LOC_LOCAL_STATUS

esp_ble_mesh_gen_user_properties_status_cb_t user_properties_status
    ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTIES_STATUS

esp_ble_mesh_gen_user_property_status_cb_t user_property_status
    ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTY_STATUS

esp_ble_mesh_gen_admin_properties_status_cb_t admin_properties_status
    ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTIES_STATUS

```

esp_ble_mesh_gen_admin_property_status_cb_t **admin_property_status**
ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTY_STATUS

esp_ble_mesh_gen_manufacturer_properties_status_cb_t **manufacturer_properties_status**
ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTIES_STATUS

esp_ble_mesh_gen_manufacturer_property_status_cb_t **manufacturer_property_status**
ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTY_STATUS

esp_ble_mesh_gen_client_properties_status_cb_t **client_properties_status**
ESP_BLE_MESH_MODEL_OP_GEN_CLIENT_PROPERTIES_STATUS

union esp_ble_mesh_generic_server_state_change_t
#include <esp_ble_mesh_generic_model_api.h> Generic Server Model state change value union.

Public Members

esp_ble_mesh_state_change_gen_onoff_set_t **onoff_set**
The rcv_op in ctx can be used to decide which state is changed. Generic OnOff Set

esp_ble_mesh_state_change_gen_level_set_t **level_set**
Generic Level Set

esp_ble_mesh_state_change_gen_delta_set_t **delta_set**
Generic Delta Set

esp_ble_mesh_state_change_gen_move_set_t **move_set**
Generic Move Set

esp_ble_mesh_state_change_gen_def_trans_time_set_t **def_trans_time_set**
Generic Default Transition Time Set

esp_ble_mesh_state_change_gen_onpowerup_set_t **onpowerup_set**
Generic OnPowerUp Set

esp_ble_mesh_state_change_gen_power_level_set_t **power_level_set**
Generic Power Level Set

esp_ble_mesh_state_change_gen_power_default_set_t **power_default_set**
Generic Power Default Set

esp_ble_mesh_state_change_gen_power_range_set_t **power_range_set**
Generic Power Range Set

esp_ble_mesh_state_change_gen_loc_global_set_t **loc_global_set**
Generic Location Global Set

esp_ble_mesh_state_change_gen_loc_local_set_t **loc_local_set**
Generic Location Local Set

esp_ble_mesh_state_change_gen_user_property_set_t **user_property_set**
Generic User Property Set

esp_ble_mesh_state_change_gen_admin_property_set_t **admin_property_set**
Generic Admin Property Set

esp_ble_mesh_state_change_gen_manu_property_set_t **manu_property_set**
Generic Manufacturer Property Set

union esp_ble_mesh_generic_server_rcv_get_msg_t
#include <esp_ble_mesh_generic_model_api.h> Generic Server Model received get message union.

Public Members

esp_ble_mesh_server_rcv_gen_user_property_get_t **user_property**
Generic User Property Get

esp_ble_mesh_server_rcv_gen_admin_property_get_t **admin_property**
Generic Admin Property Get

esp_ble_mesh_server_rcv_gen_manufacturer_property_get_t **manu_property**
Generic Manufacturer Property Get

esp_ble_mesh_server_rcv_gen_client_properties_get_t **client_properties**
Generic Client Properties Get

union esp_ble_mesh_generic_server_rcv_set_msg_t
#include <esp_ble_mesh_generic_model_api.h> Generic Server Model received set message union.

Public Members

esp_ble_mesh_server_rcv_gen_onoff_set_t **onoff**
Generic OnOff Set/Generic OnOff Set Unack

esp_ble_mesh_server_rcv_gen_level_set_t **level**
Generic Level Set/Generic Level Set Unack

esp_ble_mesh_server_rcv_gen_delta_set_t **delta**
Generic Delta Set/Generic Delta Set Unack

esp_ble_mesh_server_rcv_gen_move_set_t **move**
Generic Move Set/Generic Move Set Unack

esp_ble_mesh_server_rcv_gen_def_trans_time_set_t **def_trans_time**
Generic Default Transition Time Set/Generic Default Transition Time Set Unack

esp_ble_mesh_server_rcv_gen_onpowerup_set_t **onpowerup**
Generic OnPowerUp Set/Generic OnPowerUp Set Unack

esp_ble_mesh_server_rcv_gen_power_level_set_t **power_level**
Generic Power Level Set/Generic Power Level Set Unack

esp_ble_mesh_server_rcv_gen_power_default_set_t **power_default**
Generic Power Default Set/Generic Power Default Set Unack

esp_ble_mesh_server_rcv_gen_power_range_set_t **power_range**
Generic Power Range Set/Generic Power Range Set Unack

esp_ble_mesh_server_rcv_gen_loc_global_set_t **location_global**
Generic Location Global Set/Generic Location Global Set Unack

esp_ble_mesh_server_rcv_gen_loc_local_set_t **location_local**
Generic Location Local Set/Generic Location Local Set Unack

esp_ble_mesh_server_rcv_gen_user_property_set_t **user_property**
Generic User Property Set/Generic User Property Set Unack

esp_ble_mesh_server_rcv_gen_admin_property_set_t **admin_property**
Generic Admin Property Set/Generic Admin Property Set Unack

esp_ble_mesh_server_rcv_gen_manufacturer_property_set_t **manu_property**
Generic Manufacturer Property Set/Generic Manufacturer Property Set Unack

union esp_ble_mesh_generic_server_cb_value_t
#include <esp_ble_mesh_generic_model_api.h> Generic Server Model callback value union.

Public Members

esp_ble_mesh_generic_server_state_change_t **state_change**
ESP_BLE_MESH_GENERIC_SERVER_STATE_CHANGE_EVT

esp_ble_mesh_generic_server_rcv_get_msg_t **get**
ESP_BLE_MESH_GENERIC_SERVER_RECV_GET_MSG_EVT

esp_ble_mesh_generic_server_rcv_set_msg_t **set**
ESP_BLE_MESH_GENERIC_SERVER_RECV_SET_MSG_EVT

Structures

struct esp_ble_mesh_gen_onoff_set_t
Bluetooth Mesh Generic Client Model Get and Set parameters structure.
Parameters of Generic OnOff Set.

Public Members

bool **op_en**
Indicate if optional parameters are included

uint8_t **onoff**
Target value of Generic OnOff state

uint8_t **tid**
Transaction ID

uint8_t **trans_time**
Time to complete state transition (optional)

uint8_t **delay**
Indicate message execution delay (C.1)

struct esp_ble_mesh_gen_level_set_t
Parameters of Generic Level Set.

Public Members

bool **op_en**
Indicate if optional parameters are included

int16_t **level**
Target value of Generic Level state

uint8_t **tid**
Transaction ID

uint8_t **trans_time**
Time to complete state transition (optional)

uint8_t **delay**
Indicate message execution delay (C.1)

struct esp_ble_mesh_gen_delta_set_t
Parameters of Generic Delta Set.

Public Members

bool **op_en**
Indicate if optional parameters are included

int32_t **level**
Delta change of Generic Level state

uint8_t **tid**
Transaction ID

uint8_t **trans_time**
Time to complete state transition (optional)

`uint8_t delay`
Indicate message execution delay (C.1)

struct esp_ble_mesh_gen_move_set_t
Parameters of Generic Move Set.

Public Members

`bool op_en`
Indicate if optional parameters are included

`int16_t delta_level`
Delta Level step to calculate Move speed for Generic Level state

`uint8_t tid`
Transaction ID

`uint8_t trans_time`
Time to complete state transition (optional)

`uint8_t delay`
Indicate message execution delay (C.1)

struct esp_ble_mesh_gen_def_trans_time_set_t
Parameter of Generic Default Transition Time Set.

Public Members

`uint8_t trans_time`
The value of the Generic Default Transition Time state

struct esp_ble_mesh_gen_onpowerup_set_t
Parameter of Generic OnPowerUp Set.

Public Members

`uint8_t onpowerup`
The value of the Generic OnPowerUp state

struct esp_ble_mesh_gen_power_level_set_t
Parameters of Generic Power Level Set.

Public Members

`bool op_en`
Indicate if optional parameters are included

`uint16_t power`
Target value of Generic Power Actual state

`uint8_t tid`
Transaction ID

`uint8_t trans_time`
Time to complete state transition (optional)

`uint8_t delay`
Indicate message execution delay (C.1)

struct esp_ble_mesh_gen_power_default_set_t
Parameter of Generic Power Default Set.

Public Members**uint16_t power**

The value of the Generic Power Default state

struct esp_ble_mesh_gen_power_range_set_t

Parameters of Generic Power Range Set.

Public Members**uint16_t range_min**

Value of Range Min field of Generic Power Range state

uint16_t range_max

Value of Range Max field of Generic Power Range state

struct esp_ble_mesh_gen_loc_global_set_t

Parameters of Generic Location Global Set.

Public Members**int32_t global_latitude**

Global Coordinates (Latitude)

int32_t global_longitude

Global Coordinates (Longitude)

int16_t global_altitude

Global Altitude

struct esp_ble_mesh_gen_loc_local_set_t

Parameters of Generic Location Local Set.

Public Members**int16_t local_north**

Local Coordinates (North)

int16_t local_east

Local Coordinates (East)

int16_t local_altitude

Local Altitude

uint8_t floor_number

Floor Number

uint16_t uncertainty

Uncertainty

struct esp_ble_mesh_gen_user_property_get_t

Parameter of Generic User Property Get.

Public Members**uint16_t property_id**

Property ID identifying a Generic User Property

struct esp_ble_mesh_gen_user_property_set_t

Parameters of Generic User Property Set.

Public Members

`uint16_t property_id`
Property ID identifying a Generic User Property

`struct net_buf_simple *property_value`
Raw value for the User Property

`struct esp_ble_mesh_gen_admin_property_get_t`
Parameter of Generic Admin Property Get.

Public Members

`uint16_t property_id`
Property ID identifying a Generic Admin Property

`struct esp_ble_mesh_gen_admin_property_set_t`
Parameters of Generic Admin Property Set.

Public Members

`uint16_t property_id`
Property ID identifying a Generic Admin Property

`uint8_t user_access`
Enumeration indicating user access

`struct net_buf_simple *property_value`
Raw value for the Admin Property

`struct esp_ble_mesh_gen_manufacturer_property_get_t`
Parameter of Generic Manufacturer Property Get.

Public Members

`uint16_t property_id`
Property ID identifying a Generic Manufacturer Property

`struct esp_ble_mesh_gen_manufacturer_property_set_t`
Parameters of Generic Manufacturer Property Set.

Public Members

`uint16_t property_id`
Property ID identifying a Generic Manufacturer Property

`uint8_t user_access`
Enumeration indicating user access

`struct esp_ble_mesh_gen_client_properties_get_t`
Parameter of Generic Client Properties Get.

Public Members

`uint16_t property_id`
A starting Client Property ID present within an element

struct esp_ble_mesh_gen_onoff_status_cb_t

Bluetooth Mesh Generic Client Model Get and Set callback parameters structure.

Parameters of Generic OnOff Status.

Public Members

bool **op_en**

Indicate if optional parameters are included

uint8_t **present_onoff**

Current value of Generic OnOff state

uint8_t **target_onoff**

Target value of Generic OnOff state (optional)

uint8_t **remain_time**

Time to complete state transition (C.1)

struct esp_ble_mesh_gen_level_status_cb_t

Parameters of Generic Level Status.

Public Members

bool **op_en**

Indicate if optional parameters are included

int16_t **present_level**

Current value of Generic Level state

int16_t **target_level**

Target value of the Generic Level state (optional)

uint8_t **remain_time**

Time to complete state transition (C.1)

struct esp_ble_mesh_gen_def_trans_time_status_cb_t

Parameter of Generic Default Transition Time Status.

Public Members

uint8_t **trans_time**

The value of the Generic Default Transition Time state

struct esp_ble_mesh_gen_onpowerup_status_cb_t

Parameter of Generic OnPowerUp Status.

Public Members

uint8_t **onpowerup**

The value of the Generic OnPowerUp state

struct esp_ble_mesh_gen_power_level_status_cb_t

Parameters of Generic Power Level Status.

Public Members

bool **op_en**

Indicate if optional parameters are included

`uint16_t present_power`
Current value of Generic Power Actual state

`uint16_t target_power`
Target value of Generic Power Actual state (optional)

`uint8_t remain_time`
Time to complete state transition (C.1)

`struct esp_ble_mesh_gen_power_last_status_cb_t`
Parameter of Generic Power Last Status.

Public Members

`uint16_t power`
The value of the Generic Power Last state

`struct esp_ble_mesh_gen_power_default_status_cb_t`
Parameter of Generic Power Default Status.

Public Members

`uint16_t power`
The value of the Generic Default Last state

`struct esp_ble_mesh_gen_power_range_status_cb_t`
Parameters of Generic Power Range Status.

Public Members

`uint8_t status_code`
Status Code for the request message

`uint16_t range_min`
Value of Range Min field of Generic Power Range state

`uint16_t range_max`
Value of Range Max field of Generic Power Range state

`struct esp_ble_mesh_gen_battery_status_cb_t`
Parameters of Generic Battery Status.

Public Members

`uint32_t battery_level : 8`
Value of Generic Battery Level state

`uint32_t time_to_discharge : 24`
Value of Generic Battery Time to Discharge state

`uint32_t time_to_charge : 24`
Value of Generic Battery Time to Charge state

`uint32_t flags : 8`
Value of Generic Battery Flags state

`struct esp_ble_mesh_gen_loc_global_status_cb_t`
Parameters of Generic Location Global Status.

Public Members

`int32_t global_latitude`
Global Coordinates (Latitude)

`int32_t global_longitude`
Global Coordinates (Longitude)

`int16_t global_altitude`
Global Altitude

struct esp_ble_mesh_gen_loc_local_status_cb_t
Parameters of Generic Location Local Status.

Public Members

`int16_t local_north`
Local Coordinates (North)

`int16_t local_east`
Local Coordinates (East)

`int16_t local_altitude`
Local Altitude

`uint8_t floor_number`
Floor Number

`uint16_t uncertainty`
Uncertainty

struct esp_ble_mesh_gen_user_properties_status_cb_t
Parameter of Generic User Properties Status.

Public Members

struct net_buf_simple *property_ids
Buffer contains a sequence of N User Property IDs

struct esp_ble_mesh_gen_user_property_status_cb_t
Parameters of Generic User Property Status.

Public Members

`bool op_en`
Indicate if optional parameters are included

`uint16_t property_id`
Property ID identifying a Generic User Property

`uint8_t user_access`
Enumeration indicating user access (optional)

struct net_buf_simple *property_value
Raw value for the User Property (C.1)

struct esp_ble_mesh_gen_admin_properties_status_cb_t
Parameter of Generic Admin Properties Status.

Public Members

struct net_buf_simple ***property_ids**
Buffer contains a sequence of N Admin Property IDs

struct esp_ble_mesh_gen_admin_property_status_cb_t
Parameters of Generic Admin Property Status.

Public Members

bool **op_en**
Indicate if optional parameters are included

uint16_t **property_id**
Property ID identifying a Generic Admin Property

uint8_t **user_access**
Enumeration indicating user access (optional)

struct net_buf_simple ***property_value**
Raw value for the Admin Property (C.1)

struct esp_ble_mesh_gen_manufacturer_properties_status_cb_t
Parameter of Generic Manufacturer Properties Status.

Public Members

struct net_buf_simple ***property_ids**
Buffer contains a sequence of N Manufacturer Property IDs

struct esp_ble_mesh_gen_manufacturer_property_status_cb_t
Parameters of Generic Manufacturer Property Status.

Public Members

bool **op_en**
Indicate if optional parameters are included

uint16_t **property_id**
Property ID identifying a Generic Manufacturer Property

uint8_t **user_access**
Enumeration indicating user access (optional)

struct net_buf_simple ***property_value**
Raw value for the Manufacturer Property (C.1)

struct esp_ble_mesh_gen_client_properties_status_cb_t
Parameter of Generic Client Properties Status.

Public Members

struct net_buf_simple ***property_ids**
Buffer contains a sequence of N Client Property IDs

struct esp_ble_mesh_generic_client_cb_param_t
Generic Client Model callback parameters

Public Members**int error_code**

Appropriate error code

***esp_ble_mesh_client_common_param_t* *params**

The client common parameters.

***esp_ble_mesh_gen_client_status_cb_t* status_cb**

The generic status message callback values

struct esp_ble_mesh_gen_onoff_state_t

Parameters of Generic OnOff state

Public Members**uint8_t onoff**

The present value of the Generic OnOff state

uint8_t target_onoff

The target value of the Generic OnOff state

struct esp_ble_mesh_gen_onoff_srv_t

User data of Generic OnOff Server Model

Public Members***esp_ble_mesh_model_t* *model**

Pointer to the Generic OnOff Server Model. Initialized internally.

***esp_ble_mesh_server_rsp_ctrl_t* rsp_ctrl**

Response control of the server model received messages

***esp_ble_mesh_gen_onoff_state_t* state**

Parameters of the Generic OnOff state

***esp_ble_mesh_last_msg_info_t* last**

Parameters of the last received set message

***esp_ble_mesh_state_transition_t* transition**

Parameters of state transition

struct esp_ble_mesh_gen_level_state_t

Parameters of Generic Level state

Public Members**int16_t level**

The present value of the Generic Level state

int16_t target_level

The target value of the Generic Level state

int16_t last_level

When a new transaction starts, level should be set to last_level, and use “level + incoming delta” to calculate the target level. In another word, “last_level” is used to record “level” of the last transaction, and “last_delta” is used to record the previously received delta_level value. The last value of the Generic Level state

int32_t last_delta

The last delta change of the Generic Level state

bool **move_start**
Indicate if the transition of the Generic Level state has been started

bool **positive**
Indicate if the transition is positive or negative

struct esp_ble_mesh_gen_level_srv_t
User data of Generic Level Server Model

Public Members

esp_ble_mesh_model_t ***model**
Pointer to the Generic Level Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**
Response control of the server model received messages

esp_ble_mesh_gen_level_state_t **state**
Parameters of the Generic Level state

esp_ble_mesh_last_msg_info_t **last**
Parameters of the last received set message

esp_ble_mesh_state_transition_t **transition**
Parameters of state transition

int32_t **tt_delta_level**
Delta change value of level state transition

struct esp_ble_mesh_gen_def_trans_time_state_t
Parameter of Generic Default Transition Time state

Public Members

uint8_t **trans_time**
The value of the Generic Default Transition Time state

struct esp_ble_mesh_gen_def_trans_time_srv_t
User data of Generic Default Transition Time Server Model

Public Members

esp_ble_mesh_model_t ***model**
Pointer to the Generic Default Transition Time Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**
Response control of the server model received messages

esp_ble_mesh_gen_def_trans_time_state_t **state**
Parameters of the Generic Default Transition Time state

struct esp_ble_mesh_gen_onpowerup_state_t
Parameter of Generic OnPowerUp state

Public Members

uint8_t **onpowerup**
The value of the Generic OnPowerUp state

struct esp_ble_mesh_gen_power_onoff_srv_t
User data of Generic Power OnOff Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Generic Power OnOff Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_gen_onpowerup_state_t ***state**

Parameters of the Generic OnPowerUp state

struct esp_ble_mesh_gen_power_onoff_setup_srv_t

User data of Generic Power OnOff Setup Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Generic Power OnOff Setup Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_gen_onpowerup_state_t ***state**

Parameters of the Generic OnPowerUp state

struct esp_ble_mesh_gen_power_level_state_t

Parameters of Generic Power Level state

Public Members

uint16_t **power_actual**

The present value of the Generic Power Actual state

uint16_t **target_power_actual**

The target value of the Generic Power Actual state

uint16_t **power_last**

The value of the Generic Power Last state

uint16_t **power_default**

The value of the Generic Power Default state

uint8_t **status_code**

The status code of setting Generic Power Range state

uint16_t **power_range_min**

The minimum value of the Generic Power Range state

uint16_t **power_range_max**

The maximum value of the Generic Power Range state

struct esp_ble_mesh_gen_power_level_srv_t

User data of Generic Power Level Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Generic Power Level Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_gen_power_level_state_t *state
Parameters of the Generic Power Level state

esp_ble_mesh_last_msg_info_t last
Parameters of the last received set message

esp_ble_mesh_state_transition_t transition
Parameters of state transition

int32_t tt_delta_level
Delta change value of level state transition

struct esp_ble_mesh_gen_power_level_setup_srv_t
User data of Generic Power Level Setup Server Model

Public Members

esp_ble_mesh_model_t *model
Pointer to the Generic Power Level Setup Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t rsp_ctrl
Response control of the server model received messages

esp_ble_mesh_gen_power_level_state_t *state
Parameters of the Generic Power Level state

struct esp_ble_mesh_gen_battery_state_t
Parameters of Generic Battery state

Public Members

uint32_t battery_level : 8
The value of the Generic Battery Level state

uint32_t time_to_discharge : 24
The value of the Generic Battery Time to Discharge state

uint32_t time_to_charge : 24
The value of the Generic Battery Time to Charge state

uint32_t battery_flags : 8
The value of the Generic Battery Flags state

struct esp_ble_mesh_gen_battery_srv_t
User data of Generic Battery Server Model

Public Members

esp_ble_mesh_model_t *model
Pointer to the Generic Battery Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t rsp_ctrl
Response control of the server model received messages

esp_ble_mesh_gen_battery_state_t state
Parameters of the Generic Battery state

struct esp_ble_mesh_gen_location_state_t
Parameters of Generic Location state

Public Members

`int32_t global_latitude`
The value of the Global Latitude field

`int32_t global_longitude`
The value of the Global Longitude field

`int16_t global_altitude`
The value of the Global Altitude field

`int16_t local_north`
The value of the Local North field

`int16_t local_east`
The value of the Local East field

`int16_t local_altitude`
The value of the Local Altitude field

`uint8_t floor_number`
The value of the Floor Number field

`uint16_t uncertainty`
The value of the Uncertainty field

struct esp_ble_mesh_gen_location_srv_t
User data of Generic Location Server Model

Public Members

`esp_ble_mesh_model_t *model`
Pointer to the Generic Location Server Model. Initialized internally.

`esp_ble_mesh_server_rsp_ctrl_t rsp_ctrl`
Response control of the server model received messages

`esp_ble_mesh_gen_location_state_t *state`
Parameters of the Generic Location state

struct esp_ble_mesh_gen_location_setup_srv_t
User data of Generic Location Setup Server Model

Public Members

`esp_ble_mesh_model_t *model`
Pointer to the Generic Location Setup Server Model. Initialized internally.

`esp_ble_mesh_server_rsp_ctrl_t rsp_ctrl`
Response control of the server model received messages

`esp_ble_mesh_gen_location_state_t *state`
Parameters of the Generic Location state

struct esp_ble_mesh_generic_property_t
Parameters of Generic Property states

Public Members

`uint16_t id`
The value of User/Admin/Manufacturer Property ID

uint8_t user_access
The value of User Access field

uint8_t admin_access
The value of Admin Access field

uint8_t manu_access
The value of Manufacturer Access field

struct net_buf_simple *val
The value of User/Admin/Manufacturer Property

struct esp_ble_mesh_gen_user_prop_srv_t
User data of Generic User Property Server Model

Public Members

esp_ble_mesh_model_t ***model**
Pointer to the Generic User Property Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**
Response control of the server model received messages

uint8_t property_count
Generic User Property count

esp_ble_mesh_generic_property_t ***properties**
Parameters of the Generic User Property state

struct esp_ble_mesh_gen_admin_prop_srv_t
User data of Generic Admin Property Server Model

Public Members

esp_ble_mesh_model_t ***model**
Pointer to the Generic Admin Property Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**
Response control of the server model received messages

uint8_t property_count
Generic Admin Property count

esp_ble_mesh_generic_property_t ***properties**
Parameters of the Generic Admin Property state

struct esp_ble_mesh_gen_manu_prop_srv_t
User data of Generic Manufacturer Property Server Model

Public Members

esp_ble_mesh_model_t ***model**
Pointer to the Generic Manufacturer Property Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**
Response control of the server model received messages

uint8_t property_count
Generic Manufacturer Property count

esp_ble_mesh_generic_property_t ***properties**
Parameters of the Generic Manufacturer Property state

struct esp_ble_mesh_gen_client_prop_srv_t

User data of Generic Client Property Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Generic Client Property Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

uint8_t **id_count**

Generic Client Property ID count

uint16_t ***property_ids**

Parameters of the Generic Client Property state

struct esp_ble_mesh_state_change_gen_onoff_set_t

Parameter of Generic OnOff Set state change event

Public Members

uint8_t **onoff**

The value of Generic OnOff state

struct esp_ble_mesh_state_change_gen_level_set_t

Parameter of Generic Level Set state change event

Public Members

int16_t **level**

The value of Generic Level state

struct esp_ble_mesh_state_change_gen_delta_set_t

Parameter of Generic Delta Set state change event

Public Members

int16_t **level**

The value of Generic Level state

struct esp_ble_mesh_state_change_gen_move_set_t

Parameter of Generic Move Set state change event

Public Members

int16_t **level**

The value of Generic Level state

struct esp_ble_mesh_state_change_gen_def_trans_time_set_t

Parameter of Generic Default Transition Time Set state change event

Public Members

uint8_t **trans_time**

The value of Generic Default Transition Time state

struct esp_ble_mesh_state_change_gen_onpowerup_set_t
Parameter of Generic OnPowerUp Set state change event

Public Members

uint8_t onpowerup
The value of Generic OnPowerUp state

struct esp_ble_mesh_state_change_gen_power_level_set_t
Parameter of Generic Power Level Set state change event

Public Members

uint16_t power
The value of Generic Power Actual state

struct esp_ble_mesh_state_change_gen_power_default_set_t
Parameter of Generic Power Default Set state change event

Public Members

uint16_t power
The value of Generic Power Default state

struct esp_ble_mesh_state_change_gen_power_range_set_t
Parameters of Generic Power Range Set state change event

Public Members

uint16_t range_min
The minimum value of Generic Power Range state

uint16_t range_max
The maximum value of Generic Power Range state

struct esp_ble_mesh_state_change_gen_loc_global_set_t
Parameters of Generic Location Global Set state change event

Public Members

int32_t latitude
The Global Latitude value of Generic Location state

int32_t longitude
The Global Longitude value of Generic Location state

int16_t altitude
The Global Altitude value of Generic Location state

struct esp_ble_mesh_state_change_gen_loc_local_set_t
Parameters of Generic Location Local Set state change event

Public Members

int16_t north
The Local North value of Generic Location state

int16_t east

The Local East value of Generic Location state

int16_t altitude

The Local Altitude value of Generic Location state

uint8_t floor_number

The Floor Number value of Generic Location state

uint16_t uncertainty

The Uncertainty value of Generic Location state

struct esp_ble_mesh_state_change_gen_user_property_set_t

Parameters of Generic User Property Set state change event

Public Members

uint16_t id

The property id of Generic User Property state

struct net_buf_simple *value

The property value of Generic User Property state

struct esp_ble_mesh_state_change_gen_admin_property_set_t

Parameters of Generic Admin Property Set state change event

Public Members

uint16_t id

The property id of Generic Admin Property state

uint8_t access

The property access of Generic Admin Property state

struct net_buf_simple *value

The property value of Generic Admin Property state

struct esp_ble_mesh_state_change_gen_manu_property_set_t

Parameters of Generic Manufacturer Property Set state change event

Public Members

uint16_t id

The property id of Generic Manufacturer Property state

uint8_t access

The property value of Generic Manufacturer Property state

struct esp_ble_mesh_server_rcv_gen_user_property_get_t

Context of the received Generic User Property Get message

Public Members

uint16_t property_id

Property ID identifying a Generic User Property

struct esp_ble_mesh_server_rcv_gen_admin_property_get_t

Context of the received Generic Admin Property Get message

Public Members

`uint16_t property_id`
Property ID identifying a Generic Admin Property

`struct esp_ble_mesh_server_rcv_gen_manufacturer_property_get_t`
Context of the received Generic Manufacturer Property message

Public Members

`uint16_t property_id`
Property ID identifying a Generic Manufacturer Property

`struct esp_ble_mesh_server_rcv_gen_client_properties_get_t`
Context of the received Generic Client Properties Get message

Public Members

`uint16_t property_id`
A starting Client Property ID present within an element

`struct esp_ble_mesh_server_rcv_gen_onoff_set_t`
Context of the received Generic OnOff Set message

Public Members

`bool op_en`
Indicate if optional parameters are included

`uint8_t onoff`
Target value of Generic OnOff state

`uint8_t tid`
Transaction ID

`uint8_t trans_time`
Time to complete state transition (optional)

`uint8_t delay`
Indicate message execution delay (C.1)

`struct esp_ble_mesh_server_rcv_gen_level_set_t`
Context of the received Generic Level Set message

Public Members

`bool op_en`
Indicate if optional parameters are included

`int16_t level`
Target value of Generic Level state

`uint8_t tid`
Transaction ID

`uint8_t trans_time`
Time to complete state transition (optional)

`uint8_t delay`
Indicate message execution delay (C.1)

struct esp_ble_mesh_server_rcv_gen_delta_set_t
Context of the received Generic Delta Set message

Public Members

bool **op_en**
Indicate if optional parameters are included

int32_t **delta_level**
Delta change of Generic Level state

uint8_t **tid**
Transaction ID

uint8_t **trans_time**
Time to complete state transition (optional)

uint8_t **delay**
Indicate message execution delay (C.1)

struct esp_ble_mesh_server_rcv_gen_move_set_t
Context of the received Generic Move Set message

Public Members

bool **op_en**
Indicate if optional parameters are included

int16_t **delta_level**
Delta Level step to calculate Move speed for Generic Level state

uint8_t **tid**
Transaction ID

uint8_t **trans_time**
Time to complete state transition (optional)

uint8_t **delay**
Indicate message execution delay (C.1)

struct esp_ble_mesh_server_rcv_gen_def_trans_time_set_t
Context of the received Generic Default Transition Time Set message

Public Members

uint8_t **trans_time**
The value of the Generic Default Transition Time state

struct esp_ble_mesh_server_rcv_gen_onpowerup_set_t
Context of the received Generic OnPowerUp Set message

Public Members

uint8_t **onpowerup**
The value of the Generic OnPowerUp state

struct esp_ble_mesh_server_rcv_gen_power_level_set_t
Context of the received Generic Power Level Set message

Public Members

bool **op_en**
Indicate if optional parameters are included

uint16_t **power**
Target value of Generic Power Actual state

uint8_t **tid**
Transaction ID

uint8_t **trans_time**
Time to complete state transition (optional)

uint8_t **delay**
Indicate message execution delay (C.1)

struct esp_ble_mesh_server_rcv_gen_power_default_set_t
Context of the received Generic Power Default Set message

Public Members

uint16_t **power**
The value of the Generic Power Default state

struct esp_ble_mesh_server_rcv_gen_power_range_set_t
Context of the received Generic Power Range Set message

Public Members

uint16_t **range_min**
Value of Range Min field of Generic Power Range state

uint16_t **range_max**
Value of Range Max field of Generic Power Range state

struct esp_ble_mesh_server_rcv_gen_loc_global_set_t
Context of the received Generic Location Global Set message

Public Members

int32_t **global_latitude**
Global Coordinates (Latitude)

int32_t **global_longitude**
Global Coordinates (Longitude)

int16_t **global_altitude**
Global Altitude

struct esp_ble_mesh_server_rcv_gen_loc_local_set_t
Context of the received Generic Location Local Set message

Public Members

int16_t **local_north**
Local Coordinates (North)

int16_t **local_east**
Local Coordinates (East)

`int16_t local_altitude`
Local Altitude

`uint8_t floor_number`
Floor Number

`uint16_t uncertainty`
Uncertainty

struct esp_ble_mesh_server_rcv_gen_user_property_set_t
Context of the received Generic User Property Set message

Public Members

`uint16_t property_id`
Property ID identifying a Generic User Property

struct net_buf_simple *property_value
Raw value for the User Property

struct esp_ble_mesh_server_rcv_gen_admin_property_set_t
Context of the received Generic Admin Property Set message

Public Members

`uint16_t property_id`
Property ID identifying a Generic Admin Property

`uint8_t user_access`
Enumeration indicating user access

struct net_buf_simple *property_value
Raw value for the Admin Property

struct esp_ble_mesh_server_rcv_gen_manufacturer_property_set_t
Context of the received Generic Manufacturer Property Set message

Public Members

`uint16_t property_id`
Property ID identifying a Generic Manufacturer Property

`uint8_t user_access`
Enumeration indicating user access

struct esp_ble_mesh_generic_server_cb_param_t
Generic Server Model callback parameters

Public Members

`esp_ble_mesh_model_t *model`
Pointer to Generic Server Models

`esp_ble_mesh_msg_ctx_t ctx`
Context of the received messages

`esp_ble_mesh_generic_server_cb_value_t value`
Value of the received Generic Messages

Macros**ESP_BLE_MESH_MODEL_GEN_ONOFF_CLI** (cli_pub, cli_data)

Define a new Generic OnOff Client Model.

Note This API needs to be called for each element on which the application needs to have a Generic OnOff Client Model.**Return** New Generic OnOff Client Model instance.**Parameters**

- cli_pub: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- cli_data: Pointer to the unique struct *esp_ble_mesh_client_t*.

ESP_BLE_MESH_MODEL_GEN_LEVEL_CLI (cli_pub, cli_data)

Define a new Generic Level Client Model.

Note This API needs to be called for each element on which the application needs to have a Generic Level Client Model.**Return** New Generic Level Client Model instance.**Parameters**

- cli_pub: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- cli_data: Pointer to the unique struct *esp_ble_mesh_client_t*.

ESP_BLE_MESH_MODEL_GEN_DEF_TRANS_TIME_CLI (cli_pub, cli_data)

Define a new Generic Default Transition Time Client Model.

Note This API needs to be called for each element on which the application needs to have a Generic Default Transition Time Client Model.**Return** New Generic Default Transition Time Client Model instance.**Parameters**

- cli_pub: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- cli_data: Pointer to the unique struct *esp_ble_mesh_client_t*.

ESP_BLE_MESH_MODEL_GEN_POWER_ONOFF_CLI (cli_pub, cli_data)

Define a new Generic Power OnOff Client Model.

Note This API needs to be called for each element on which the application needs to have a Generic Power OnOff Client Model.**Return** New Generic Power OnOff Client Model instance.**Parameters**

- cli_pub: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- cli_data: Pointer to the unique struct *esp_ble_mesh_client_t*.

ESP_BLE_MESH_MODEL_GEN_POWER_LEVEL_CLI (cli_pub, cli_data)

Define a new Generic Power Level Client Model.

Note This API needs to be called for each element on which the application needs to have a Generic Power Level Client Model.**Return** New Generic Power Level Client Model instance.**Parameters**

- cli_pub: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- cli_data: Pointer to the unique struct *esp_ble_mesh_client_t*.

ESP_BLE_MESH_MODEL_GEN_BATTERY_CLI (cli_pub, cli_data)

Define a new Generic Battery Client Model.

Note This API needs to be called for each element on which the application needs to have a Generic Battery Client Model.**Return** New Generic Battery Client Model instance.**Parameters**

- cli_pub: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- cli_data: Pointer to the unique struct *esp_ble_mesh_client_t*.

ESP_BLE_MESH_MODEL_GEN_LOCATION_CLI (cli_pub, cli_data)

Define a new Generic Location Client Model.

Note This API needs to be called for each element on which the application needs to have a Generic Location Client Model.

Return New Generic Location Client Model instance.

Parameters

- `cli_pub`: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- `cli_data`: Pointer to the unique struct *esp_ble_mesh_client_t*.

ESP_BLE_MESH_MODEL_GEN_PROPERTY_CLI (`cli_pub`, `cli_data`)

Define a new Generic Property Client Model.

Note This API needs to be called for each element on which the application needs to have a Generic Property Client Model.

Return New Generic Location Client Model instance.

Parameters

- `cli_pub`: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- `cli_data`: Pointer to the unique struct *esp_ble_mesh_client_t*.

ESP_BLE_MESH_MODEL_GEN_ONOFF_SRV (`srv_pub`, `srv_data`)

Generic Server Models related context.

Define a new Generic OnOff Server Model.

Note 1. The Generic OnOff Server Model is a root model.

1. This model shall support model publication and model subscription.

Return New Generic OnOff Server Model instance.

Parameters

- `srv_pub`: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- `srv_data`: Pointer to the unique struct *esp_ble_mesh_gen_onoff_srv_t*.

ESP_BLE_MESH_MODEL_GEN_LEVEL_SRV (`srv_pub`, `srv_data`)

Define a new Generic Level Server Model.

Note 1. The Generic Level Server Model is a root model.

1. This model shall support model publication and model subscription.

Return New Generic Level Server Model instance.

Parameters

- `srv_pub`: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- `srv_data`: Pointer to the unique struct *esp_ble_mesh_gen_level_srv_t*.

ESP_BLE_MESH_MODEL_GEN_DEF_TRANS_TIME_SRV (`srv_pub`, `srv_data`)

Define a new Generic Default Transition Time Server Model.

Note 1. The Generic Default Transition Time Server Model is a root model.

1. This model shall support model publication and model subscription.

Return New Generic Default Transition Time Server Model instance.

Parameters

- `srv_pub`: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- `srv_data`: Pointer to the unique struct *esp_ble_mesh_gen_def_trans_time_srv_t*.

ESP_BLE_MESH_MODEL_GEN_POWER_ONOFF_SRV (`srv_pub`, `srv_data`)

Define a new Generic Power OnOff Server Model.

Note 1. The Generic Power OnOff Server model extends the Generic OnOff Server model. When this model is present on an element, the corresponding Generic Power OnOff Setup Server model shall also be present.

1. This model may be used to represent a variety of devices that do not fit any of the model descriptions that have been defined but support the generic properties of On/Off.
2. This model shall support model publication and model subscription.

Return New Generic Power OnOff Server Model instance.

Parameters

- `srv_pub`: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- `srv_data`: Pointer to the unique struct *esp_ble_mesh_gen_power_onoff_srv_t*.

ESP_BLE_MESH_MODEL_GEN_POWER_ONOFF_SETUP_SRV (srv_pub, srv_data)

Define a new Generic Power OnOff Setup Server Model.

Note 1. The Generic Power OnOff Setup Server model extends the Generic Power OnOff Server model and the Generic Default Transition Time Server model.

1. This model shall support model subscription.

Return New Generic Power OnOff Setup Server Model instance.

Parameters

- *srv_pub*: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- *srv_data*: Pointer to the unique struct *esp_ble_mesh_gen_power_onoff_setup_srv_t*.

ESP_BLE_MESH_MODEL_GEN_POWER_LEVEL_SRV (srv_pub, srv_data)

Define a new Generic Power Level Server Model.

Note 1. The Generic Power Level Server model extends the Generic Power OnOff Server model and the Generic Level Server model. When this model is present on an Element, the corresponding Generic Power Level Setup Server model shall also be present.

1. This model shall support model publication and model subscription.

Return New Generic Power Level Server Model instance.

Parameters

- *srv_pub*: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- *srv_data*: Pointer to the unique struct *esp_ble_mesh_gen_power_level_srv_t*.

ESP_BLE_MESH_MODEL_GEN_POWER_LEVEL_SETUP_SRV (srv_pub, srv_data)

Define a new Generic Power Level Setup Server Model.

Note 1. The Generic Power Level Setup Server model extends the Generic Power Level Server model and the Generic Power OnOff Setup Server model.

1. This model shall support model subscription.

Return New Generic Power Level Setup Server Model instance.

Parameters

- *srv_pub*: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- *srv_data*: Pointer to the unique struct *esp_ble_mesh_gen_power_level_setup_srv_t*.

ESP_BLE_MESH_MODEL_GEN_BATTERY_SRV (srv_pub, srv_data)

Define a new Generic Battery Server Model.

Note 1. The Generic Battery Server Model is a root model.

1. This model shall support model publication and model subscription.
2. The model may be used to represent an element that is powered by a battery.

Return New Generic Battery Server Model instance.

Parameters

- *srv_pub*: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- *srv_data*: Pointer to the unique struct *esp_ble_mesh_gen_battery_srv_t*.

ESP_BLE_MESH_MODEL_GEN_LOCATION_SRV (srv_pub, srv_data)

Define a new Generic Location Server Model.

Note 1. The Generic Location Server model is a root model. When this model is present on an Element, the corresponding Generic Location Setup Server model shall also be present.

1. This model shall support model publication and model subscription.
2. The model may be used to represent an element that knows its location (global or local).

Return New Generic Location Server Model instance.

Parameters

- *srv_pub*: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- *srv_data*: Pointer to the unique struct *esp_ble_mesh_gen_location_srv_t*.

ESP_BLE_MESH_MODEL_GEN_LOCATION_SETUP_SRV (srv_pub, srv_data)

Define a new Generic Location Setup Server Model.

Note 1. The Generic Location Setup Server model extends the Generic Location Server model.

1. This model shall support model subscription.

Return New Generic Location Setup Server Model instance.

Parameters

- `srv_pub`: Pointer to the unique struct `esp_ble_mesh_model_pub_t`.
- `srv_data`: Pointer to the unique struct `esp_ble_mesh_gen_location_setup_srv_t`.

ESP_BLE_MESH_MODEL_GEN_USER_PROP_SRV (`srv_pub`, `srv_data`)

Define a new Generic User Property Server Model.

Note 1. The Generic User Property Server model is a root model.

1. This model shall support model publication and model subscription.

Return New Generic User Property Server Model instance.

Parameters

- `srv_pub`: Pointer to the unique struct `esp_ble_mesh_model_pub_t`.
- `srv_data`: Pointer to the unique struct `esp_ble_mesh_gen_user_prop_srv_t`.

ESP_BLE_MESH_MODEL_GEN_ADMIN_PROP_SRV (`srv_pub`, `srv_data`)

Define a new Generic Admin Property Server Model.

Note 1. The Generic Admin Property Server model extends the Generic User Property Server model.

1. This model shall support model publication and model subscription.

Return New Generic Admin Property Server Model instance.

Parameters

- `srv_pub`: Pointer to the unique struct `esp_ble_mesh_model_pub_t`.
- `srv_data`: Pointer to the unique struct `esp_ble_mesh_gen_admin_prop_srv_t`.

ESP_BLE_MESH_MODEL_GEN_MANUFACTURER_PROP_SRV (`srv_pub`, `srv_data`)

Define a new Generic Manufacturer Property Server Model.

Note 1. The Generic Manufacturer Property Server model extends the Generic User Property Server model.

1. This model shall support model publication and model subscription.

Return New Generic Manufacturer Property Server Model instance.

Parameters

- `srv_pub`: Pointer to the unique struct `esp_ble_mesh_model_pub_t`.
- `srv_data`: Pointer to the unique struct `esp_ble_mesh_gen_manu_prop_srv_t`.

ESP_BLE_MESH_MODEL_GEN_CLIENT_PROP_SRV (`srv_pub`, `srv_data`)

Define a new Generic User Property Server Model.

Note 1. The Generic Client Property Server model is a root model.

1. This model shall support model publication and model subscription.

Return New Generic Client Property Server Model instance.

Parameters

- `srv_pub`: Pointer to the unique struct `esp_ble_mesh_model_pub_t`.
- `srv_data`: Pointer to the unique struct `esp_ble_mesh_gen_client_prop_srv_t`.

Type Definitions

```
typedef void (*esp_ble_mesh_generic_client_cb_t) (esp_ble_mesh_generic_client_cb_event_t
                                                event,
                                                esp_ble_mesh_generic_client_cb_param_t
                                                *param)
```

Bluetooth Mesh Generic Client Model function.

Generic Client Model callback function type

Parameters

- `event`: Event type
- `param`: Pointer to callback parameter

```
typedef void (*esp_ble_mesh_generic_server_cb_t) (esp_ble_mesh_generic_server_cb_event_t
                                                event,
                                                esp_ble_mesh_generic_server_cb_param_t
                                                *param)
```

Bluetooth Mesh Generic Server Model function.

Generic Server Model callback function type

Parameters

- event: Event type
- param: Pointer to callback parameter

Enumerations

enum esp_ble_mesh_generic_client_cb_event_t

This enum value is the event of Generic Client Model

Values:

ESP_BLE_MESH_GENERIC_CLIENT_GET_STATE_EVT

ESP_BLE_MESH_GENERIC_CLIENT_SET_STATE_EVT

ESP_BLE_MESH_GENERIC_CLIENT_PUBLISH_EVT

ESP_BLE_MESH_GENERIC_CLIENT_TIMEOUT_EVT

ESP_BLE_MESH_GENERIC_CLIENT_EVT_MAX

enum esp_ble_mesh_gen_user_prop_access_t

This enum value is the access value of Generic User Property

Values:

ESP_BLE_MESH_GEN_USER_ACCESS_PROHIBIT

ESP_BLE_MESH_GEN_USER_ACCESS_READ

ESP_BLE_MESH_GEN_USER_ACCESS_WRITE

ESP_BLE_MESH_GEN_USER_ACCESS_READ_WRITE

enum esp_ble_mesh_gen_admin_prop_access_t

This enum value is the access value of Generic Admin Property

Values:

ESP_BLE_MESH_GEN_ADMIN_NOT_USER_PROP

ESP_BLE_MESH_GEN_ADMIN_ACCESS_READ

ESP_BLE_MESH_GEN_ADMIN_ACCESS_WRITE

ESP_BLE_MESH_GEN_ADMIN_ACCESS_READ_WRITE

enum esp_ble_mesh_gen_manu_prop_access_t

This enum value is the access value of Generic Manufacturer Property

Values:

ESP_BLE_MESH_GEN_MANU_NOT_USER_PROP

ESP_BLE_MESH_GEN_MANU_ACCESS_READ

enum esp_ble_mesh_generic_server_cb_event_t

This enum value is the event of Generic Server Model

Values:

ESP_BLE_MESH_GENERIC_SERVER_STATE_CHANGE_EVT

1. When get_auto_rsp is set to ESP_BLE_MESH_SERVER_AUTO_RSP, no event will be callback to the application layer when Generic Get messages are received.
2. When set_auto_rsp is set to ESP_BLE_MESH_SERVER_AUTO_RSP, this event will be callback to the application layer when Generic Set/Set Unack messages are received.

ESP_BLE_MESH_GENERIC_SERVER_RECV_GET_MSG_EVT

When get_auto_rsp is set to ESP_BLE_MESH_SERVER_RSP_BY_APP, this event will be callback to the application layer when Generic Get messages are received.

ESP_BLE_MESH_GENERIC_SERVER_RECV_SET_MSG_EVT

When `set_auto_rsp` is set to `ESP_BLE_MESH_SERVER_RSP_BY_APP`, this event will be callback to the application layer when Generic Set/Set Unack messages are received.

ESP_BLE_MESH_GENERIC_SERVER_EVT_MAX

Generic Client/Server Models

Header File

- [bt/esp_ble_mesh/api/models/include/esp_ble_mesh_health_model_api.h](#)

Functions

esp_err_t **esp_ble_mesh_register_health_client_callback** (*esp_ble_mesh_health_client_cb_t* callback)

Register BLE Mesh Health Model callback, the callback will report Health Client & Server Model events.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] `callback`: Pointer to the callback function.

esp_err_t **esp_ble_mesh_register_health_server_callback** (*esp_ble_mesh_health_server_cb_t* callback)

Register BLE Mesh Health Server Model callback.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] `callback`: Pointer to the callback function.

esp_err_t **esp_ble_mesh_health_client_get_state** (*esp_ble_mesh_client_common_param_t* *params, *esp_ble_mesh_health_client_get_state_t* *get_state)

This function is called to get the Health Server states using the Health Client Model get messages.

Note If you want to find the opcodes and corresponding meanings accepted by this API, please refer to `esp_ble_mesh_opcode_health_client_get_t` in `esp_ble_mesh_defs.h`

Return ESP_OK on success or error code otherwise.

Parameters

- [in] `params`: Pointer to BLE Mesh common client parameters.
- [in] `get_state`: Pointer to a union, each kind of opcode corresponds to one structure inside. Shall not be set to NULL.

esp_err_t **esp_ble_mesh_health_client_set_state** (*esp_ble_mesh_client_common_param_t* *params, *esp_ble_mesh_health_client_set_state_t* *set_state)

This function is called to set the Health Server states using the Health Client Model set messages.

Note If you want to find the opcodes and corresponding meanings accepted by this API, please refer to `esp_ble_mesh_opcode_health_client_set_t` in `esp_ble_mesh_defs.h`

Return ESP_OK on success or error code otherwise.

Parameters

- [in] `params`: Pointer to BLE Mesh common client parameters.
- [in] `set_state`: Pointer to a union, each kind of opcode corresponds to one structure inside. Shall not be set to NULL.

esp_err_t **esp_ble_mesh_health_server_fault_update** (*esp_ble_mesh_elem_t* *element)

This function is called by the Health Server Model to update the context of its Health Current status.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] `element`: The element to which the Health Server Model belongs.

Unions**union esp_ble_mesh_health_client_get_state_t**

#include <esp_ble_mesh_health_model_api.h> For ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_GET ESP_BLE_MESH_MODEL_OP_ATTENTION_GET ESP_BLE_MESH_MODEL_OP_HEALTH_PERIOD_GET the get_state parameter in the esp_ble_mesh_health_client_get_state function should not be set to NULL.

Public Members*esp_ble_mesh_health_fault_get_t* **fault_get**

For ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_GET.

union esp_ble_mesh_health_client_set_state_t

#include <esp_ble_mesh_health_model_api.h> For ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_CLEAR ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_CLEAR_UNACK ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_TEST ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_TEST_UNACK ESP_BLE_MESH_MODEL_OP_HEALTH_PERIOD_SET ESP_BLE_MESH_MODEL_OP_HEALTH_PERIOD_SET_UNACK ESP_BLE_MESH_MODEL_OP_ATTENTION_SET ESP_BLE_MESH_MODEL_OP_ATTENTION_SET_UNACK the set_state parameter in the esp_ble_mesh_health_client_set_state function should not be set to NULL.

Public Members*esp_ble_mesh_health_attention_set_t* **attention_set**

For ESP_BLE_MESH_MODEL_OP_ATTENTION_SET or ESP_BLE_MESH_MODEL_OP_ATTENTION_SET_UNACK.

esp_ble_mesh_health_period_set_t **period_set**

For ESP_BLE_MESH_MODEL_OP_HEALTH_PERIOD_SET or ESP_BLE_MESH_MODEL_OP_HEALTH_PERIOD_SET_UNACK.

esp_ble_mesh_health_fault_test_t **fault_test**

For ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_TEST or ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_TEST_UNACK.

esp_ble_mesh_health_fault_clear_t **fault_clear**

For ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_CLEAR or ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_CLEAR_UNACK.

union esp_ble_mesh_health_client_common_cb_param_t

#include <esp_ble_mesh_health_model_api.h> Health Client Model received message union.

Public Members*esp_ble_mesh_health_current_status_cb_t* **current_status**

The health current status value

esp_ble_mesh_health_fault_status_cb_t **fault_status**

The health fault status value

esp_ble_mesh_health_period_status_cb_t **period_status**

The health period status value

esp_ble_mesh_health_attention_status_cb_t **attention_status**

The health attention status value

union esp_ble_mesh_health_server_cb_param_t

#include <esp_ble_mesh_health_model_api.h> Health Server Model callback parameters union.

Public Members*esp_ble_mesh_health_fault_update_comp_cb_t* **fault_update_comp**

ESP_BLE_MESH_HEALTH_SERVER_FAULT_UPDATE_COMP_EVT

esp_ble_mesh_health_fault_clear_cb_t **fault_clear**
ESP_BLE_MESH_HEALTH_SERVER_FAULT_CLEAR_EVT

esp_ble_mesh_health_fault_test_cb_t **fault_test**
ESP_BLE_MESH_HEALTH_SERVER_FAULT_TEST_EVT

esp_ble_mesh_health_attention_on_cb_t **attention_on**
ESP_BLE_MESH_HEALTH_SERVER_ATTENTION_ON_EVT

esp_ble_mesh_health_attention_off_cb_t **attention_off**
ESP_BLE_MESH_HEALTH_SERVER_ATTENTION_OFF_EVT

Structures

struct esp_ble_mesh_health_srv_cb_t
ESP BLE Mesh Health Server callback

Public Members

esp_ble_mesh_cb_t **fault_clear**
Clear health registered faults. Initialized by the stack.

esp_ble_mesh_cb_t **fault_test**
Run a specific health test. Initialized by the stack.

esp_ble_mesh_cb_t **attention_on**
Health attention on callback. Initialized by the stack.

esp_ble_mesh_cb_t **attention_off**
Health attention off callback. Initialized by the stack.

struct esp_ble_mesh_health_test_t
ESP BLE Mesh Health Server test Context

Public Members

uint8_t id_count
Number of Health self-test ID

const uint8_t *test_ids
Array of Health self-test IDs

uint16_t company_id
Company ID used to identify the Health Fault state

uint8_t prev_test_id
Current test ID of the health fault test

uint8_t current_faults[ESP_BLE_MESH_HEALTH_FAULT_ARRAY_SIZE]
Array of current faults

uint8_t registered_faults[ESP_BLE_MESH_HEALTH_FAULT_ARRAY_SIZE]
Array of registered faults

struct esp_ble_mesh_health_srv_t
ESP BLE Mesh Health Server Model Context

Public Members

esp_ble_mesh_model_t ***model**
Pointer to Health Server Model

esp_ble_mesh_health_srv_cb_t **health_cb**
Health callback struct

struct `k_delayed_work_attention_timer`
Attention Timer state

bool `attention_timer_start`
Attention Timer start flag

esp_ble_mesh_health_test_t `health_test`
Health Server fault test

struct `esp_ble_mesh_health_fault_get_t`
Parameter of Health Fault Get

Public Members

uint16_t `company_id`
Bluetooth assigned 16-bit Company ID

struct `esp_ble_mesh_health_attention_set_t`
Parameter of Health Attention Set

Public Members

uint8_t `attention`
Value of the Attention Timer state

struct `esp_ble_mesh_health_period_set_t`
Parameter of Health Period Set

Public Members

uint8_t `fast_period_divisor`
Divider for the Publish Period

struct `esp_ble_mesh_health_fault_test_t`
Parameter of Health Fault Test

Public Members

uint16_t `company_id`
Bluetooth assigned 16-bit Company ID

uint8_t `test_id`
ID of a specific test to be performed

struct `esp_ble_mesh_health_fault_clear_t`
Parameter of Health Fault Clear

Public Members

uint16_t `company_id`
Bluetooth assigned 16-bit Company ID

struct `esp_ble_mesh_health_current_status_cb_t`
Parameters of Health Current Status

Public Members

`uint8_t test_id`
ID of a most recently performed test

`uint16_t company_id`
Bluetooth assigned 16-bit Company ID

`struct net_buf_simple *fault_array`
FaultArray field contains a sequence of 1-octet fault values

`struct esp_ble_mesh_health_fault_status_cb_t`
Parameters of Health Fault Status

Public Members

`uint8_t test_id`
ID of a most recently performed test

`uint16_t company_id`
Bluetooth assigned 16-bit Company ID

`struct net_buf_simple *fault_array`
FaultArray field contains a sequence of 1-octet fault values

`struct esp_ble_mesh_health_period_status_cb_t`
Parameter of Health Period Status

Public Members

`uint8_t fast_period_divisor`
Divider for the Publish Period

`struct esp_ble_mesh_health_attention_status_cb_t`
Parameter of Health Attention Status

Public Members

`uint8_t attention`
Value of the Attention Timer state

`struct esp_ble_mesh_health_client_cb_param_t`
Health Client Model callback parameters

Public Members

`int error_code`
Appropriate error code

`esp_ble_mesh_client_common_param_t *params`
The client common parameters.

`esp_ble_mesh_health_client_common_cb_param_t status_cb`
The health message status callback values

`struct esp_ble_mesh_health_fault_update_comp_cb_t`
Parameter of publishing Health Current Status completion event

Public Members

int **error_code**
The result of publishing Health Current Status

esp_ble_mesh_elem_t ***element**
Pointer to the element which contains the Health Server Model

struct esp_ble_mesh_health_fault_clear_cb_t
Parameters of Health Fault Clear event

Public Members

esp_ble_mesh_model_t ***model**
Pointer to the Health Server Model

uint16_t **company_id**
Bluetooth assigned 16-bit Company ID

struct esp_ble_mesh_health_fault_test_cb_t
Parameters of Health Fault Test event

Public Members

esp_ble_mesh_model_t ***model**
Pointer to the Health Server Model

uint8_t **test_id**
ID of a specific test to be performed

uint16_t **company_id**
Bluetooth assigned 16-bit Company ID

struct esp_ble_mesh_health_attention_on_cb_t
Parameter of Health Attention On event

Public Members

esp_ble_mesh_model_t ***model**
Pointer to the Health Server Model

uint8_t **time**
Duration of attention timer on (in seconds)

struct esp_ble_mesh_health_attention_off_cb_t
Parameter of Health Attention Off event

Public Members

esp_ble_mesh_model_t ***model**
Pointer to the Health Server Model

Macros

ESP_BLE_MESH_MODEL_HEALTH_SRV (srv, pub)
Define a new Health Server Model.

Note The Health Server Model can only be included by a Primary Element.

Return New Health Server Model instance.

Parameters

- `srv`: Pointer to the unique struct `esp_ble_mesh_health_srv_t`.
- `pub`: Pointer to the unique struct `esp_ble_mesh_model_pub_t`.

ESP_BLE_MESH_MODEL_HEALTH_CLI (`cli_data`)

Define a new Health Client Model.

Note This API needs to be called for each element on which the application needs to have a Health Client Model.

Return New Health Client Model instance.

Parameters

- `cli_data`: Pointer to the unique struct `esp_ble_mesh_client_t`.

ESP_BLE_MESH_HEALTH_PUB_DEFINE (`_name`, `_max`, `_role`)

A helper to define a health publication context

Parameters

- `_name`: Name given to the publication context variable.
- `_max`: Maximum number of faults the element can have.
- `_role`: Role of the device which contains the model.

ESP_BLE_MESH_HEALTH_STANDARD_TEST

SIG identifier of Health Fault Test. 0x01 ~ 0xFF: Vendor Specific Test.

ESP_BLE_MESH_NO_FAULT

Fault values of Health Fault Test. 0x33 ~ 0x7F: Reserved for Future Use. 0x80 ~ 0xFF: Vendor Specific Warning/Error.

ESP_BLE_MESH_BATTERY_LOW_WARNING

ESP_BLE_MESH_BATTERY_LOW_ERROR

ESP_BLE_MESH_SUPPLY_VOLTAGE_TOO_LOW_WARNING

ESP_BLE_MESH_SUPPLY_VOLTAGE_TOO_LOW_ERROR

ESP_BLE_MESH_SUPPLY_VOLTAGE_TOO_HIGH_WARNING

ESP_BLE_MESH_SUPPLY_VOLTAGE_TOO_HIGH_ERROR

ESP_BLE_MESH_POWER_SUPPLY_INTERRUPTED_WARNING

ESP_BLE_MESH_POWER_SUPPLY_INTERRUPTED_ERROR

ESP_BLE_MESH_NO_LOAD_WARNING

ESP_BLE_MESH_NO_LOAD_ERROR

ESP_BLE_MESH_OVERLOAD_WARNING

ESP_BLE_MESH_OVERLOAD_ERROR

ESP_BLE_MESH_OVERHEAT_WARNING

ESP_BLE_MESH_OVERHEAT_ERROR

ESP_BLE_MESH_CONDENSATION_WARNING

ESP_BLE_MESH_CONDENSATION_ERROR

ESP_BLE_MESH_VIBRATION_WARNING

ESP_BLE_MESH_VIBRATION_ERROR

ESP_BLE_MESH_CONFIGURATION_WARNING

ESP_BLE_MESH_CONFIGURATION_ERROR

ESP_BLE_MESH_ELEMENT_NOT_CALIBRATED_WARNING

ESP_BLE_MESH_ELEMENT_NOT_CALIBRATED_ERROR

ESP_BLE_MESH_MEMORY_WARNING

ESP_BLE_MESH_MEMORY_ERROR

ESP_BLE_MESH_SELF_TEST_WARNING
ESP_BLE_MESH_SELF_TEST_ERROR
ESP_BLE_MESH_INPUT_TOO_LOW_WARNING
ESP_BLE_MESH_INPUT_TOO_LOW_ERROR
ESP_BLE_MESH_INPUT_TOO_HIGH_WARNING
ESP_BLE_MESH_INPUT_TOO_HIGH_ERROR
ESP_BLE_MESH_INPUT_NO_CHANGE_WARNING
ESP_BLE_MESH_INPUT_NO_CHANGE_ERROR
ESP_BLE_MESH_ACTUATOR_BLOCKED_WARNING
ESP_BLE_MESH_ACTUATOR_BLOCKED_ERROR
ESP_BLE_MESH_HOUSING_OPENED_WARNING
ESP_BLE_MESH_HOUSING_OPENED_ERROR
ESP_BLE_MESH_TAMPER_WARNING
ESP_BLE_MESH_TAMPER_ERROR
ESP_BLE_MESH_DEVICE_MOVED_WARNING
ESP_BLE_MESH_DEVICE_MOVED_ERROR
ESP_BLE_MESH_DEVICE_DROPPED_WARNING
ESP_BLE_MESH_DEVICE_DROPPED_ERROR
ESP_BLE_MESH_OVERFLOW_WARNING
ESP_BLE_MESH_OVERFLOW_ERROR
ESP_BLE_MESH_EMPTY_WARNING
ESP_BLE_MESH_EMPTY_ERROR
ESP_BLE_MESH_INTERNAL_BUS_WARNING
ESP_BLE_MESH_INTERNAL_BUS_ERROR
ESP_BLE_MESH_MECHANISM_JAMMED_WARNING
ESP_BLE_MESH_MECHANISM_JAMMED_ERROR
ESP_BLE_MESH_HEALTH_FAULT_ARRAY_SIZE

Type Definitions

typedef void (***esp_ble_mesh_health_client_cb_t**) (*esp_ble_mesh_health_client_cb_event_t* event, *esp_ble_mesh_health_client_cb_param_t* *param)

Bluetooth Mesh Health Client and Server Model function.

Health Client Model callback function type

Parameters

- event: Event type
- param: Pointer to callback parameter

typedef void (***esp_ble_mesh_health_server_cb_t**) (*esp_ble_mesh_health_server_cb_event_t* event, *esp_ble_mesh_health_server_cb_param_t* *param)

Health Server Model callback function type.

Parameters

- event: Event type

- param: Pointer to callback parameter

Enumerations

enum esp_ble_mesh_health_client_cb_event_t

This enum value is the event of Health Client Model

Values:

ESP_BLE_MESH_HEALTH_CLIENT_GET_STATE_EVT

ESP_BLE_MESH_HEALTH_CLIENT_SET_STATE_EVT

ESP_BLE_MESH_HEALTH_CLIENT_PUBLISH_EVT

ESP_BLE_MESH_HEALTH_CLIENT_TIMEOUT_EVT

ESP_BLE_MESH_HEALTH_CLIENT_EVT_MAX

enum esp_ble_mesh_health_server_cb_event_t

This enum value is the event of Health Server Model

Values:

ESP_BLE_MESH_HEALTH_SERVER_FAULT_UPDATE_COMP_EVT

ESP_BLE_MESH_HEALTH_SERVER_FAULT_CLEAR_EVT

ESP_BLE_MESH_HEALTH_SERVER_FAULT_TEST_EVT

ESP_BLE_MESH_HEALTH_SERVER_ATTENTION_ON_EVT

ESP_BLE_MESH_HEALTH_SERVER_ATTENTION_OFF_EVT

ESP_BLE_MESH_HEALTH_SERVER_EVT_MAX

Sensor Client/Server Models

Header File

- [bt/esp_ble_mesh/api/models/include/esp_ble_mesh_lighting_model_api.h](#)

Functions

esp_err_t esp_ble_mesh_register_light_client_callback (*esp_ble_mesh_light_client_cb_t* callback)

Register BLE Mesh Light Client Model callback.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] callback: pointer to the callback function.

esp_err_t esp_ble_mesh_light_client_get_state (*esp_ble_mesh_client_common_param_t* *params, *esp_ble_mesh_light_client_get_state_t* *get_state)

Get the value of Light Server Model states using the Light Client Model get messages.

Note If you want to know the opcodes and corresponding meanings accepted by this API, please refer to esp_ble_mesh_light_message_opcode_t in esp_ble_mesh_defs.h

Return ESP_OK on success or error code otherwise.

Parameters

- [in] params: Pointer to BLE Mesh common client parameters.
- [in] get_state: Pointer of light get message value. Shall not be set to NULL.

esp_err_t esp_ble_mesh_light_client_set_state (*esp_ble_mesh_client_common_param_t* *params, *esp_ble_mesh_light_client_set_state_t* *set_state)

Set the value of Light Server Model states using the Light Client Model set messages.

Note If you want to know the opcodes and corresponding meanings accepted by this API, please refer to `esp_ble_mesh_light_message_opcode_t` in `esp_ble_mesh_defs.h`

Return ESP_OK on success or error code otherwise.

Parameters

- [in] `params`: Pointer to BLE Mesh common client parameters.
- [in] `set_state`: Pointer of light set message value. Shall not be set to NULL.

`esp_err_t esp_ble_mesh_register_lighting_server_callback` (`esp_ble_mesh_lighting_server_cb_t` callback)

Register BLE Mesh Lighting Server Model callback.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] `callback`: Pointer to the callback function.

Unions

union `esp_ble_mesh_light_client_get_state_t`

`#include <esp_ble_mesh_lighting_model_api.h>` Lighting Client Model get message union.

Public Members

`esp_ble_mesh_light_lc_property_get_t` `lc_property_get`

For ESP_BLE_MESH_MODEL_OP_LIGHT_LC_PROPERTY_GET

union `esp_ble_mesh_light_client_set_state_t`

`#include <esp_ble_mesh_lighting_model_api.h>` Lighting Client Model set message union.

Public Members

`esp_ble_mesh_light_lightness_set_t` `lightness_set`

For ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_SET &
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_SET_UNACK

`esp_ble_mesh_light_lightness_linear_set_t` `lightness_linear_set`

For ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_LINEAR_SET &
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_LINEAR_SET_UNACK

`esp_ble_mesh_light_lightness_default_set_t` `lightness_default_set`

For ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_DEFAULT_SET &
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_DEFAULT_SET_UNACK

`esp_ble_mesh_light_lightness_range_set_t` `lightness_range_set`

For ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_RANGE_SET &
ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_RANGE_SET_UNACK

`esp_ble_mesh_light_ctl_set_t` `ctl_set`

For ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_SET & ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_SET_UNACK

`esp_ble_mesh_light_ctl_temperature_set_t` `ctl_temperature_set`

For ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_SET &
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_SET_UNACK

`esp_ble_mesh_light_ctl_temperature_range_set_t` `ctl_temperature_range_set`

For ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_RANGE_SET &
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_RANGE_SET_UNACK

`esp_ble_mesh_light_ctl_default_set_t` `ctl_default_set`

For ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_DEFAULT_SET &
ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_DEFAULT_SET_UNACK

`esp_ble_mesh_light_hsl_set_t` `hsl_set`

For ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_SET & ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_SET_UNACK


```

esp_ble_mesh_light_hsl_hue_set_t hsl_hue_set
    For ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_HUE_SET & ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_HUE_SET_UNACK

esp_ble_mesh_light_hsl_saturation_set_t hsl_saturation_set
    For ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_SATURATION_SET &
    ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_SATURATION_SET_UNACK

esp_ble_mesh_light_hsl_default_set_t hsl_default_set
    For ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_DEFAULT_SET &
    ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_DEFAULT_SET_UNACK

esp_ble_mesh_light_hsl_range_set_t hsl_range_set
    For ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_RANGE_SET &
    ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_RANGE_SET_UNACK

esp_ble_mesh_light_xyl_set_t xyl_set
    For ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_SET & ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_SET_UNACK

esp_ble_mesh_light_xyl_default_set_t xyl_default_set
    For ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_DEFAULT_SET &
    ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_DEFAULT_SET_UNACK

esp_ble_mesh_light_xyl_range_set_t xyl_range_set
    For ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_RANGE_SET &
    ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_RANGE_SET_UNACK

esp_ble_mesh_light_lc_mode_set_t lc_mode_set
    For ESP_BLE_MESH_MODEL_OP_LIGHT_LC_MODE_SET &
    ESP_BLE_MESH_MODEL_OP_LIGHT_LC_MODE_SET_UNACK

esp_ble_mesh_light_lc_om_set_t lc_om_set
    For ESP_BLE_MESH_MODEL_OP_LIGHT_LC_OM_SET & ESP_BLE_MESH_MODEL_OP_LIGHT_LC_OM_SET_UNACK

esp_ble_mesh_light_lc_light_onoff_set_t lc_light_onoff_set
    For ESP_BLE_MESH_MODEL_OP_LIGHT_LC_LIGHT_ONOFF_SET &
    ESP_BLE_MESH_MODEL_OP_LIGHT_LC_LIGHT_ONOFF_SET_UNACK

esp_ble_mesh_light_lc_property_set_t lc_property_set
    For ESP_BLE_MESH_MODEL_OP_LIGHT_LC_PROPERTY_SET &
    ESP_BLE_MESH_MODEL_OP_LIGHT_LC_PROPERTY_SET_UNACK

```

```

union esp_ble_mesh_light_client_status_cb_t

```

```

    #include <esp_ble_mesh_lighting_model_api.h> Lighting Client Model received message union.

```

Public Members

```

esp_ble_mesh_light_lightness_status_cb_t lightness_status
    For ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_STATUS

esp_ble_mesh_light_lightness_linear_status_cb_t lightness_linear_status
    For ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_LINEAR_STATUS

esp_ble_mesh_light_lightness_last_status_cb_t lightness_last_status
    For ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_LAST_STATUS

esp_ble_mesh_light_lightness_default_status_cb_t lightness_default_status
    For ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_DEFAULT_STATUS

esp_ble_mesh_light_lightness_range_status_cb_t lightness_range_status
    For ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_RANGE_STATUS

esp_ble_mesh_light_ctl_status_cb_t ctl_status
    For ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_STATUS

esp_ble_mesh_light_ctl_temperature_status_cb_t ctl_temperature_status
    For ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_STATUS

```

esp_ble_mesh_light_ctl_temperature_range_status_cb_t **ctl_temperature_range_status**
For ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_TEMPERATURE_RANGE_STATUS

esp_ble_mesh_light_ctl_default_status_cb_t **ctl_default_status**
For ESP_BLE_MESH_MODEL_OP_LIGHT_CTL_DEFAULT_STATUS

esp_ble_mesh_light_hsl_status_cb_t **hsl_status**
For ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_STATUS

esp_ble_mesh_light_hsl_target_status_cb_t **hsl_target_status**
For ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_TARGET_STATUS

esp_ble_mesh_light_hsl_hue_status_cb_t **hsl_hue_status**
For ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_HUE_STATUS

esp_ble_mesh_light_hsl_saturation_status_cb_t **hsl_saturation_status**
For ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_SATURATION_STATUS

esp_ble_mesh_light_hsl_default_status_cb_t **hsl_default_status**
For ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_DEFAULT_STATUS

esp_ble_mesh_light_hsl_range_status_cb_t **hsl_range_status**
For ESP_BLE_MESH_MODEL_OP_LIGHT_HSL_RANGE_STATUS

esp_ble_mesh_light_xyl_status_cb_t **xyl_status**
For ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_STATUS

esp_ble_mesh_light_xyl_target_status_cb_t **xyl_target_status**
For ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_TARGET_STATUS

esp_ble_mesh_light_xyl_default_status_cb_t **xyl_default_status**
For ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_DEFAULT_STATUS

esp_ble_mesh_light_xyl_range_status_cb_t **xyl_range_status**
For ESP_BLE_MESH_MODEL_OP_LIGHT_XYL_RANGE_STATUS

esp_ble_mesh_light_lc_mode_status_cb_t **lc_mode_status**
For ESP_BLE_MESH_MODEL_OP_LIGHT_LC_MODE_STATUS

esp_ble_mesh_light_lc_om_status_cb_t **lc_om_status**
For ESP_BLE_MESH_MODEL_OP_LIGHT_LC_OM_STATUS

esp_ble_mesh_light_lc_light_onoff_status_cb_t **lc_light_onoff_status**
For ESP_BLE_MESH_MODEL_OP_LIGHT_LC_LIGHT_ONOFF_STATUS

esp_ble_mesh_light_lc_property_status_cb_t **lc_property_status**
For ESP_BLE_MESH_MODEL_OP_LIGHT_LC_PROPERTY_STATUS

union esp_ble_mesh_lighting_server_state_change_t
#include <esp_ble_mesh_lighting_model_api.h> Lighting Server Model state change value union.

Public Members

esp_ble_mesh_state_change_light_lightness_set_t **lightness_set**
The recv_op in ctx can be used to decide which state is changed. Light Lightness Set

esp_ble_mesh_state_change_light_lightness_linear_set_t **lightness_linear_set**
Light Lightness Linear Set

esp_ble_mesh_state_change_light_lightness_default_set_t **lightness_default_set**
Light Lightness Default Set

esp_ble_mesh_state_change_light_lightness_range_set_t **lightness_range_set**
Light Lightness Range Set

esp_ble_mesh_state_change_light_ctl_set_t **ctl_set**
Light CTL Set

esp_ble_mesh_state_change_light_ctl_temperature_set_t **ctl_temp_set**
Light CTL Temperature Set

esp_ble_mesh_state_change_light_ctl_temperature_range_set_t **ctl_temp_range_set**
Light CTL Temperature Range Set

esp_ble_mesh_state_change_light_ctl_default_set_t **ctl_default_set**
Light CTL Default Set

esp_ble_mesh_state_change_light_hsl_set_t **hsl_set**
Light HSL Set

esp_ble_mesh_state_change_light_hsl_hue_set_t **hsl_hue_set**
Light HSL Hue Set

esp_ble_mesh_state_change_light_hsl_saturation_set_t **hsl_saturation_set**
Light HSL Saturation Set

esp_ble_mesh_state_change_light_hsl_default_set_t **hsl_default_set**
Light HSL Default Set

esp_ble_mesh_state_change_light_hsl_range_set_t **hsl_range_set**
Light HSL Range Set

esp_ble_mesh_state_change_light_xyl_set_t **xyl_set**
Light xyL Set

esp_ble_mesh_state_change_light_xyl_default_set_t **xyl_default_set**
Light xyL Default Set

esp_ble_mesh_state_change_light_xyl_range_set_t **xyl_range_set**
Light xyL Range Set

esp_ble_mesh_state_change_light_lc_mode_set_t **lc_mode_set**
Light LC Mode Set

esp_ble_mesh_state_change_light_lc_om_set_t **lc_om_set**
Light LC Occupancy Mode Set

esp_ble_mesh_state_change_light_lc_light_onoff_set_t **lc_light_onoff_set**
Light LC Light OnOff Set

esp_ble_mesh_state_change_light_lc_property_set_t **lc_property_set**
Light LC Property Set

esp_ble_mesh_state_change_sensor_status_t **sensor_status**
Sensor Status

union esp_ble_mesh_lighting_server_recv_get_msg_t
#include <esp_ble_mesh_lighting_model_api.h> Lighting Server Model received get message union.

Public Members

esp_ble_mesh_server_recv_light_lc_property_get_t **lc_property**
Light LC Property Get

union esp_ble_mesh_lighting_server_recv_set_msg_t
#include <esp_ble_mesh_lighting_model_api.h> Lighting Server Model received set message union.

Public Members

esp_ble_mesh_server_recv_light_lightness_set_t **lightness**
Light Lightness Set/Light Lightness Set Unack

esp_ble_mesh_server_recv_light_lightness_linear_set_t **lightness_linear**
Light Lightness Linear Set/Light Lightness Linear Set Unack

esp_ble_mesh_server_rcv_light_lightness_default_set_t **lightness_default**
Light Lightness Default Set/Light Lightness Default Set Unack

esp_ble_mesh_server_rcv_light_lightness_range_set_t **lightness_range**
Light Lightness Range Set/Light Lightness Range Set Unack

esp_ble_mesh_server_rcv_light_ctl_set_t **ctl**
Light CTL Set/Light CTL Set Unack

esp_ble_mesh_server_rcv_light_ctl_temperature_set_t **ctl_temp**
Light CTL Temperature Set/Light CTL Temperature Set Unack

esp_ble_mesh_server_rcv_light_ctl_temperature_range_set_t **ctl_temp_range**
Light CTL Temperature Range Set/Light CTL Temperature Range Set Unack

esp_ble_mesh_server_rcv_light_ctl_default_set_t **ctl_default**
Light CTL Default Set/Light CTL Default Set Unack

esp_ble_mesh_server_rcv_light_hsl_set_t **hsl**
Light HSL Set/Light HSL Set Unack

esp_ble_mesh_server_rcv_light_hsl_hue_set_t **hsl_hue**
Light HSL Hue Set/Light HSL Hue Set Unack

esp_ble_mesh_server_rcv_light_hsl_saturation_set_t **hsl_saturation**
Light HSL Saturation Set/Light HSL Saturation Set Unack

esp_ble_mesh_server_rcv_light_hsl_default_set_t **hsl_default**
Light HSL Default Set/Light HSL Default Set Unack

esp_ble_mesh_server_rcv_light_hsl_range_set_t **hsl_range**
Light HSL Range Set/Light HSL Range Set Unack

esp_ble_mesh_server_rcv_light_xyl_set_t **xyl**
Light xyL Set/Light xyL Set Unack

esp_ble_mesh_server_rcv_light_xyl_default_set_t **xyl_default**
Light xyL Default Set/Light xyL Default Set Unack

esp_ble_mesh_server_rcv_light_xyl_range_set_t **xyl_range**
Light xyL Range Set/Light xyL Range Set Unack

esp_ble_mesh_server_rcv_light_lc_mode_set_t **lc_mode**
Light LC Mode Set/Light LC Mode Set Unack

esp_ble_mesh_server_rcv_light_lc_om_set_t **lc_om**
Light LC OM Set/Light LC OM Set Unack

esp_ble_mesh_server_rcv_light_lc_light_onoff_set_t **lc_light_onoff**
Light LC Light OnOff Set/Light LC Light OnOff Set Unack

esp_ble_mesh_server_rcv_light_lc_property_set_t **lc_property**
Light LC Property Set/Light LC Property Set Unack

union esp_ble_mesh_lighting_server_rcv_status_msg_t
#include <esp_ble_mesh_lighting_model_api.h> Lighting Server Model received status message union.

Public Members

esp_ble_mesh_server_rcv_sensor_status_t **sensor_status**
Sensor Status

union esp_ble_mesh_lighting_server_cb_value_t
#include <esp_ble_mesh_lighting_model_api.h> Lighting Server Model callback value union.

Public Members

esp_ble_mesh_lighting_server_state_change_t **state_change**
ESP_BLE_MESH_LIGHTING_SERVER_STATE_CHANGE_EVT

esp_ble_mesh_lighting_server_rcv_get_msg_t **get**
ESP_BLE_MESH_LIGHTING_SERVER_RECV_GET_MSG_EVT

esp_ble_mesh_lighting_server_rcv_set_msg_t **set**
ESP_BLE_MESH_LIGHTING_SERVER_RECV_SET_MSG_EVT

esp_ble_mesh_lighting_server_rcv_status_msg_t **status**
ESP_BLE_MESH_LIGHTING_SERVER_RECV_STATUS_MSG_EVT

Structures

struct esp_ble_mesh_light_lightness_set_t
Bluetooth Mesh Light Lightness Client Model Get and Set parameters structure.
Parameters of Light Lightness Set

Public Members

bool **op_en**
Indicate if optional parameters are included

uint16_t **lightness**
Target value of light lightness actual state

uint8_t **tid**
Transaction ID

uint8_t **trans_time**
Time to complete state transition (optional)

uint8_t **delay**
Indicate message execution delay (C.1)

struct esp_ble_mesh_light_lightness_linear_set_t
Parameters of Light Lightness Linear Set

Public Members

bool **op_en**
Indicate if optional parameters are included

uint16_t **lightness**
Target value of light lightness linear state

uint8_t **tid**
Transaction ID

uint8_t **trans_time**
Time to complete state transition (optional)

uint8_t **delay**
Indicate message execution delay (C.1)

struct esp_ble_mesh_light_lightness_default_set_t
Parameter of Light Lightness Default Set

Public Members**uint16_t lightness**

The value of the Light Lightness Default state

struct esp_ble_mesh_light_lightness_range_set_t

Parameters of Light Lightness Range Set

Public Members**uint16_t range_min**

Value of range min field of light lightness range state

uint16_t range_max

Value of range max field of light lightness range state

struct esp_ble_mesh_light_ctl_set_t

Parameters of Light CTL Set

Public Members**bool op_en**

Indicate if optional parameters are included

uint16_t ctl_lightness

Target value of light ctl lightness state

uint16_t ctl_temperatrue

Target value of light ctl temperature state

int16_t ctl_delta_uv

Target value of light ctl delta UV state

uint8_t tid

Transaction ID

uint8_t trans_time

Time to complete state transition (optional)

uint8_t delay

Indicate message execution delay (C.1)

struct esp_ble_mesh_light_ctl_temperature_set_t

Parameters of Light CTL Temperature Set

Public Members**bool op_en**

Indicate if optional parameters are included

uint16_t ctl_temperatrue

Target value of light ctl temperature state

int16_t ctl_delta_uv

Target value of light ctl delta UV state

uint8_t tid

Transaction ID

uint8_t trans_time

Time to complete state transition (optional)

`uint8_t delay`

Indicate message execution delay (C.1)

struct esp_ble_mesh_light_ctl_temperature_range_set_t

Parameters of Light CTL Temperature Range Set

Public Members

`uint16_t range_min`

Value of temperature range min field of light ctl temperature range state

`uint16_t range_max`

Value of temperature range max field of light ctl temperature range state

struct esp_ble_mesh_light_ctl_default_set_t

Parameters of Light CTL Default Set

Public Members

`uint16_t lightness`

Value of light lightness default state

`uint16_t temperature`

Value of light temperature default state

`int16_t delta_uv`

Value of light delta UV default state

struct esp_ble_mesh_light_hsl_set_t

Parameters of Light HSL Set

Public Members

`bool op_en`

Indicate if optional parameters are included

`uint16_t hsl_lightness`

Target value of light hsl lightness state

`uint16_t hsl_hue`

Target value of light hsl hue state

`uint16_t hsl_saturation`

Target value of light hsl saturation state

`uint8_t tid`

Transaction ID

`uint8_t trans_time`

Time to complete state transition (optional)

`uint8_t delay`

Indicate message execution delay (C.1)

struct esp_ble_mesh_light_hsl_hue_set_t

Parameters of Light HSL Hue Set

Public Members

`bool op_en`

Indicate if optional parameters are included

`uint16_t hue`
Target value of light hsl hue state

`uint8_t tid`
Transaction ID

`uint8_t trans_time`
Time to complete state transition (optional)

`uint8_t delay`
Indicate message execution delay (C.1)

struct esp_ble_mesh_light_hsl_saturation_set_t
Parameters of Light HSL Saturation Set

Public Members

bool `op_en`
Indicate if optional parameters are included

`uint16_t saturation`
Target value of light hsl hue state

`uint8_t tid`
Transaction ID

`uint8_t trans_time`
Time to complete state transition (optional)

`uint8_t delay`
Indicate message execution delay (C.1)

struct esp_ble_mesh_light_hsl_default_set_t
Parameters of Light HSL Default Set

Public Members

`uint16_t lightness`
Value of light lightness default state

`uint16_t hue`
Value of light hue default state

`uint16_t saturation`
Value of light saturation default state

struct esp_ble_mesh_light_hsl_range_set_t
Parameters of Light HSL Range Set

Public Members

`uint16_t hue_range_min`
Value of hue range min field of light hsl hue range state

`uint16_t hue_range_max`
Value of hue range max field of light hsl hue range state

`uint16_t saturation_range_min`
Value of saturation range min field of light hsl saturation range state

`uint16_t saturation_range_max`
Value of saturation range max field of light hsl saturation range state

struct esp_ble_mesh_light_xyl_set_t

Parameters of Light xyL Set

Public Members**bool op_en**

Indicate whether optional parameters included

uint16_t xyl_lightness

The target value of the Light xyL Lightness state

uint16_t xyl_x

The target value of the Light xyL x state

uint16_t xyl_y

The target value of the Light xyL y state

uint8_t tid

Transaction Identifier

uint8_t trans_time

Time to complete state transition (optional)

uint8_t delay

Indicate message execution delay (C.1)

struct esp_ble_mesh_light_xyl_default_set_t

Parameters of Light xyL Default Set

Public Members**uint16_t lightness**

The value of the Light Lightness Default state

uint16_t xyl_x

The value of the Light xyL x Default state

uint16_t xyl_y

The value of the Light xyL y Default state

struct esp_ble_mesh_light_xyl_range_set_t

Parameters of Light xyL Range Set

Public Members**uint16_t xyl_x_range_min**

The value of the xyL x Range Min field of the Light xyL x Range state

uint16_t xyl_x_range_max

The value of the xyL x Range Max field of the Light xyL x Range state

uint16_t xyl_y_range_min

The value of the xyL y Range Min field of the Light xyL y Range state

uint16_t xyl_y_range_max

The value of the xyL y Range Max field of the Light xyL y Range state

struct esp_ble_mesh_light_lc_mode_set_t

Parameter of Light LC Mode Set

Public Members**uint8_t mode**

The target value of the Light LC Mode state

struct esp_ble_mesh_light_lc_om_set_t

Parameter of Light LC OM Set

Public Members**uint8_t mode**

The target value of the Light LC Occupancy Mode state

struct esp_ble_mesh_light_lc_light_onoff_set_t

Parameters of Light LC Light OnOff Set

Public Members**bool op_en**

Indicate whether optional parameters included

uint8_t light_onoff

The target value of the Light LC Light OnOff state

uint8_t tid

Transaction Identifier

uint8_t trans_time

Time to complete state transition (optional)

uint8_t delay

Indicate message execution delay (C.1)

struct esp_ble_mesh_light_lc_property_get_t

Parameter of Light LC Property Get

Public Members**uint16_t property_id**

Property ID identifying a Light LC Property

struct esp_ble_mesh_light_lc_property_set_t

Parameters of Light LC Property Set

Public Members**uint16_t property_id**

Property ID identifying a Light LC Property

struct net_buf_simple *property_value

Raw value for the Light LC Property

struct esp_ble_mesh_light_lightness_status_cb_t

Bluetooth Mesh Light Lightness Client Model Get and Set callback parameters structure.

Parameters of Light Lightness Status

Public Members

bool op_en
Indicate if optional parameters are included

uint16_t present_lightness
Current value of light lightness actual state

uint16_t target_lightness
Target value of light lightness actual state (optional)

uint8_t remain_time
Time to complete state transition (C.1)

struct esp_ble_mesh_light_lightness_linear_status_cb_t
Parameters of Light Lightness Linear Status

Public Members

bool op_en
Indicate if optional parameters are included

uint16_t present_lightness
Current value of light lightness linear state

uint16_t target_lightness
Target value of light lightness linear state (optional)

uint8_t remain_time
Time to complete state transition (C.1)

struct esp_ble_mesh_light_lightness_last_status_cb_t
Parameter of Light Lightness Last Status

Public Members

uint16_t lightness
The value of the Light Lightness Last state

struct esp_ble_mesh_light_lightness_default_status_cb_t
Parameter of Light Lightness Default Status

Public Members

uint16_t lightness
The value of the Light Lightness default State

struct esp_ble_mesh_light_lightness_range_status_cb_t
Parameters of Light Lightness Range Status

Public Members

uint8_t status_code
Status Code for the request message

uint16_t range_min
Value of range min field of light lightness range state

uint16_t range_max
Value of range max field of light lightness range state

struct esp_ble_mesh_light_ctl_status_cb_t
Parameters of Light CTL Status

Public Members

bool op_en
Indicate if optional parameters are included

uint16_t present_ctl_lightness
Current value of light ctl lightness state

uint16_t present_ctl_temperature
Current value of light ctl temperature state

uint16_t target_ctl_lightness
Target value of light ctl lightness state (optional)

uint16_t target_ctl_temperature
Target value of light ctl temperature state (C.1)

uint8_t remain_time
Time to complete state transition (C.1)

struct esp_ble_mesh_light_ctl_temperature_status_cb_t
Parameters of Light CTL Temperature Status

Public Members

bool op_en
Indicate if optional parameters are included

uint16_t present_ctl_temperature
Current value of light ctl temperature state

uint16_t present_ctl_delta_uv
Current value of light ctl delta UV state

uint16_t target_ctl_temperature
Target value of light ctl temperature state (optional)

uint16_t target_ctl_delta_uv
Target value of light ctl delta UV state (C.1)

uint8_t remain_time
Time to complete state transition (C.1)

struct esp_ble_mesh_light_ctl_temperature_range_status_cb_t
Parameters of Light CTL Temperature Range Status

Public Members

uint8_t status_code
Status code for the request message

uint16_t range_min
Value of temperature range min field of light ctl temperature range state

uint16_t range_max
Value of temperature range max field of light ctl temperature range state

struct esp_ble_mesh_light_ctl_default_status_cb_t
Parameters of Light CTL Default Status

Public Members

`uint16_t lightness`
Value of light lightness default state

`uint16_t temperature`
Value of light temperature default state

`int16_t delta_uv`
Value of light delta UV default state

`struct esp_ble_mesh_light_hsl_status_cb_t`
Parameters of Light HSL Status

Public Members

`bool op_en`
Indicate if optional parameters are included

`uint16_t hsl_lightness`
Current value of light hsl lightness state

`uint16_t hsl_hue`
Current value of light hsl hue state

`uint16_t hsl_saturation`
Current value of light hsl saturation state

`uint8_t remain_time`
Time to complete state transition (optional)

`struct esp_ble_mesh_light_hsl_target_status_cb_t`
Parameters of Light HSL Target Status

Public Members

`bool op_en`
Indicate if optional parameters are included

`uint16_t hsl_lightness_target`
Target value of light hsl lightness state

`uint16_t hsl_hue_target`
Target value of light hsl hue state

`uint16_t hsl_saturation_target`
Target value of light hsl saturation state

`uint8_t remain_time`
Time to complete state transition (optional)

`struct esp_ble_mesh_light_hsl_hue_status_cb_t`
Parameters of Light HSL Hue Status

Public Members

`bool op_en`
Indicate if optional parameters are included

`uint16_t present_hue`
Current value of light hsl hue state

uint16_t target_hue
Target value of light hsl hue state (optional)

uint8_t remain_time
Time to complete state transition (C.1)

struct esp_ble_mesh_light_hsl_saturation_status_cb_t
Parameters of Light HSL Saturation Status

Public Members

bool op_en
Indicate if optional parameters are included

uint16_t present_saturation
Current value of light hsl saturation state

uint16_t target_saturation
Target value of light hsl saturation state (optional)

uint8_t remain_time
Time to complete state transition (C.1)

struct esp_ble_mesh_light_hsl_default_status_cb_t
Parameters of Light HSL Default Status

Public Members

uint16_t lightness
Value of light lightness default state

uint16_t hue
Value of light hue default state

uint16_t saturation
Value of light saturation default state

struct esp_ble_mesh_light_hsl_range_status_cb_t
Parameters of Light HSL Range Status

Public Members

uint8_t status_code
Status code for the request message

uint16_t hue_range_min
Value of hue range min field of light hsl hue range state

uint16_t hue_range_max
Value of hue range max field of light hsl hue range state

uint16_t saturation_range_min
Value of saturation range min field of light hsl saturation range state

uint16_t saturation_range_max
Value of saturation range max field of light hsl saturation range state

struct esp_ble_mesh_light_xyl_status_cb_t
Parameters of Light xyL Status

Public Members

bool op_en
Indicate whether optional parameters included

uint16_t xyl_lightness
The present value of the Light xyL Lightness state

uint16_t xyl_x
The present value of the Light xyL x state

uint16_t xyl_y
The present value of the Light xyL y state

uint8_t remain_time
Time to complete state transition (optional)

struct esp_ble_mesh_light_xyl_target_status_cb_t
Parameters of Light xyL Target Status

Public Members

bool op_en
Indicate whether optional parameters included

uint16_t target_xyl_lightness
The target value of the Light xyL Lightness state

uint16_t target_xyl_x
The target value of the Light xyL x state

uint16_t target_xyl_y
The target value of the Light xyL y state

uint8_t remain_time
Time to complete state transition (optional)

struct esp_ble_mesh_light_xyl_default_status_cb_t
Parameters of Light xyL Default Status

Public Members

uint16_t lightness
The value of the Light Lightness Default state

uint16_t xyl_x
The value of the Light xyL x Default state

uint16_t xyl_y
The value of the Light xyL y Default state

struct esp_ble_mesh_light_xyl_range_status_cb_t
Parameters of Light xyL Range Status

Public Members

uint8_t status_code
Status Code for the requesting message

uint16_t xyl_x_range_min
The value of the xyL x Range Min field of the Light xyL x Range state

`uint16_t xyl_x_range_max`

The value of the xyL x Range Max field of the Light xyL x Range state

`uint16_t xyl_y_range_min`

The value of the xyL y Range Min field of the Light xyL y Range state

`uint16_t xyl_y_range_max`

The value of the xyL y Range Max field of the Light xyL y Range state

struct esp_ble_mesh_light_lc_mode_status_cb_t

Parameter of Light LC Mode Status

Public Members

`uint8_t mode`

The present value of the Light LC Mode state

struct esp_ble_mesh_light_lc_om_status_cb_t

Parameter of Light LC OM Status

Public Members

`uint8_t mode`

The present value of the Light LC Occupancy Mode state

struct esp_ble_mesh_light_lc_light_onoff_status_cb_t

Parameters of Light LC Light OnOff Status

Public Members

`bool op_en`

Indicate whether optional parameters included

`uint8_t present_light_onoff`

The present value of the Light LC Light OnOff state

`uint8_t target_light_onoff`

The target value of the Light LC Light OnOff state (Optional)

`uint8_t remain_time`

Time to complete state transition (C.1)

struct esp_ble_mesh_light_lc_property_status_cb_t

Parameters of Light LC Property Status

Public Members

`uint16_t property_id`

Property ID identifying a Light LC Property

struct net_buf_simple *property_value

Raw value for the Light LC Property

struct esp_ble_mesh_light_client_cb_param_t

Lighting Client Model callback parameters

Public Members**int error_code**

Appropriate error code

***esp_ble_mesh_client_common_param_t* *params**

The client common parameters.

***esp_ble_mesh_light_client_status_cb_t* status_cb**

The light status message callback values

struct esp_ble_mesh_light_lightness_state_t

Parameters of Light Lightness state

Public Members**uint16_t lightness_linear**

The present value of Light Lightness Linear state

uint16_t target_lightness_linear

The target value of Light Lightness Linear state

uint16_t lightness_actual

The present value of Light Lightness Actual state

uint16_t target_lightness_actual

The target value of Light Lightness Actual state

uint16_t lightness_last

The value of Light Lightness Last state

uint16_t lightness_default

The value of Light Lightness Default state

uint8_t status_code

The status code of setting Light Lightness Range state

uint16_t lightness_range_min

The minimum value of Light Lightness Range state

uint16_t lightness_range_max

The maximum value of Light Lightness Range state

struct esp_ble_mesh_light_lightness_srv_t

User data of Light Lightness Server Model

Public Members***esp_ble_mesh_model_t* *model**

Pointer to the Lighting Lightness Server Model. Initialized internally.

***esp_ble_mesh_server_rsp_ctrl_t* rsp_ctrl**

Response control of the server model received messages

***esp_ble_mesh_light_lightness_state_t* *state**

Parameters of the Light Lightness state

***esp_ble_mesh_last_msg_info_t* last**

Parameters of the last received set message

***esp_ble_mesh_state_transition_t* actual_transition**

Parameters of state transition

***esp_ble_mesh_state_transition_t* linear_transition**

Parameters of state transition

int32_t tt_delta_lightness_actual
Delta change value of lightness actual state transition

int32_t tt_delta_lightness_linear
Delta change value of lightness linear state transition

struct esp_ble_mesh_light_lightness_setup_srv_t
User data of Light Lightness Setup Server Model

Public Members

esp_ble_mesh_model_t *model
Pointer to the Lighting Lightness Setup Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t rsp_ctrl
Response control of the server model received messages

esp_ble_mesh_light_lightness_state_t *state
Parameters of the Light Lightness state

struct esp_ble_mesh_light_ctl_state_t
Parameters of Light CTL state

Public Members

uint16_t lightness
The present value of Light CTL Lightness state

uint16_t target_lightness
The target value of Light CTL Lightness state

uint16_t temperature
The present value of Light CTL Temperature state

uint16_t target_temperature
The target value of Light CTL Temperature state

int16_t delta_uv
The present value of Light CTL Delta UV state

int16_t target_delta_uv
The target value of Light CTL Delta UV state

uint8_t status_code
The statue code of setting Light CTL Temperature Range state

uint16_t temperature_range_min
The minimum value of Light CTL Temperature Range state

uint16_t temperature_range_max
The maximum value of Light CTL Temperature Range state

uint16_t lightness_default
The value of Light Lightness Default state

uint16_t temperature_default
The value of Light CTL Temperature Default state

int16_t delta_uv_default
The value of Light CTL Delta UV Default state

struct esp_ble_mesh_light_ctl_srv_t
User data of Light CTL Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Lighting CTL Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_light_ctl_state_t ***state**

Parameters of the Light CTL state

esp_ble_mesh_last_msg_info_t **last**

Parameters of the last received set message

esp_ble_mesh_state_transition_t **transition**

Parameters of state transition

int32_t **tt_delta_lightness**

Delta change value of lightness state transition

int32_t **tt_delta_temperature**

Delta change value of temperature state transition

int32_t **tt_delta_delta_uv**

Delta change value of delta uv state transition

struct esp_ble_mesh_light_ctl_setup_srv_t

User data of Light CTL Setup Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Lighting CTL Setup Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_light_ctl_state_t ***state**

Parameters of the Light CTL state

struct esp_ble_mesh_light_ctl_temp_srv_t

User data of Light CTL Temperature Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Lighting CTL Temperature Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_light_ctl_state_t ***state**

Parameters of the Light CTL state

esp_ble_mesh_last_msg_info_t **last**

Parameters of the last received set message

esp_ble_mesh_state_transition_t **transition**

Parameters of state transition

int32_t **tt_delta_temperature**

Delta change value of temperature state transition

int32_t **tt_delta_delta_uv**

Delta change value of delta uv state transition

struct esp_ble_mesh_light_hsl_state_t
Parameters of Light HSL state

Public Members

uint16_t lightness
The present value of Light HSL Lightness state

uint16_t target_lightness
The target value of Light HSL Lightness state

uint16_t hue
The present value of Light HSL Hue state

uint16_t target_hue
The target value of Light HSL Hue state

uint16_t saturation
The present value of Light HSL Saturation state

uint16_t target_saturation
The target value of Light HSL Saturation state

uint16_t lightness_default
The value of Light Lightness Default state

uint16_t hue_default
The value of Light HSL Hue Default state

uint16_t saturation_default
The value of Light HSL Saturation Default state

uint8_t status_code
The status code of setting Light HSL Hue & Saturation Range state

uint16_t hue_range_min
The minimum value of Light HSL Hue Range state

uint16_t hue_range_max
The maximum value of Light HSL Hue Range state

uint16_t saturation_range_min
The minimum value of Light HSL Saturation state

uint16_t saturation_range_max
The maximum value of Light HSL Saturation state

struct esp_ble_mesh_light_hsl_srv_t
User data of Light HSL Server Model

Public Members

esp_ble_mesh_model_t ***model**
Pointer to the Lighting HSL Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**
Response control of the server model received messages

esp_ble_mesh_light_hsl_state_t ***state**
Parameters of the Light HSL state

esp_ble_mesh_last_msg_info_t **last**
Parameters of the last received set message

esp_ble_mesh_state_transition_t **transition**
Parameters of state transition

`int32_t tt_delta_lightness`
Delta change value of lightness state transition

`int32_t tt_delta_hue`
Delta change value of hue state transition

`int32_t tt_delta_saturation`
Delta change value of saturation state transition

struct esp_ble_mesh_light_hsl_setup_srv_t
User data of Light HSL Setup Server Model

Public Members

`esp_ble_mesh_model_t *model`
Pointer to the Lighting HSL Setup Server Model. Initialized internally.

`esp_ble_mesh_server_rsp_ctrl_t rsp_ctrl`
Response control of the server model received messages

`esp_ble_mesh_light_hsl_state_t *state`
Parameters of the Light HSL state

struct esp_ble_mesh_light_hsl_hue_srv_t
User data of Light HSL Hue Server Model

Public Members

`esp_ble_mesh_model_t *model`
Pointer to the Lighting HSL Hue Server Model. Initialized internally.

`esp_ble_mesh_server_rsp_ctrl_t rsp_ctrl`
Response control of the server model received messages

`esp_ble_mesh_light_hsl_state_t *state`
Parameters of the Light HSL state

`esp_ble_mesh_last_msg_info_t last`
Parameters of the last received set message

`esp_ble_mesh_state_transition_t transition`
Parameters of state transition

`int32_t tt_delta_hue`
Delta change value of hue state transition

struct esp_ble_mesh_light_hsl_sat_srv_t
User data of Light HSL Saturation Server Model

Public Members

`esp_ble_mesh_model_t *model`
Pointer to the Lighting HSL Saturation Server Model. Initialized internally.

`esp_ble_mesh_server_rsp_ctrl_t rsp_ctrl`
Response control of the server model received messages

`esp_ble_mesh_light_hsl_state_t *state`
Parameters of the Light HSL state

`esp_ble_mesh_last_msg_info_t last`
Parameters of the last received set message

esp_ble_mesh_state_transition_t **transition**

Parameters of state transition

int32_t **tt_delta_saturation**

Delta change value of saturation state transition

struct esp_ble_mesh_light_xyl_state_t

Parameters of Light xyL state

Public Members

uint16_t **lightness**

The present value of Light xyL Lightness state

uint16_t **target_lightness**

The target value of Light xyL Lightness state

uint16_t **x**

The present value of Light xyL x state

uint16_t **target_x**

The target value of Light xyL x state

uint16_t **y**

The present value of Light xyL y state

uint16_t **target_y**

The target value of Light xyL y state

uint16_t **lightness_default**

The value of Light Lightness Default state

uint16_t **x_default**

The value of Light xyL x Default state

uint16_t **y_default**

The value of Light xyL y Default state

uint8_t **status_code**

The status code of setting Light xyL x & y Range state

uint16_t **x_range_min**

The minimum value of Light xyL x Range state

uint16_t **x_range_max**

The maximum value of Light xyL x Range state

uint16_t **y_range_min**

The minimum value of Light xyL y Range state

uint16_t **y_range_max**

The maximum value of Light xyL y Range state

struct esp_ble_mesh_light_xyl_srv_t

User data of Light xyL Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Lighting xyL Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_light_xyl_state_t ***state**

Parameters of the Light xyL state

esp_ble_mesh_last_msg_info_t **last**

Parameters of the last received set message

esp_ble_mesh_state_transition_t **transition**

Parameters of state transition

int32_t **tt_delta_lightness**

Delta change value of lightness state transition

int32_t **tt_delta_x**

Delta change value of x state transition

int32_t **tt_delta_y**

Delta change value of y state transition

struct esp_ble_mesh_light_xyl_setup_srv_t

User data of Light xyL Setup Server Model

Public Members

esp_ble_mesh_model_t ***model**

Pointer to the Lighting xyL Setup Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**

Response control of the server model received messages

esp_ble_mesh_light_xyl_state_t ***state**

Parameters of the Light xyL state

struct esp_ble_mesh_light_lc_state_t

Parameters of Light LC states

Public Members

uint32_t **mode** : 1

0b0 The controller is turned off.

- The binding with the Light Lightness state is disabled. 0b1 The controller is turned on.
- The binding with the Light Lightness state is enabled. The value of Light LC Mode state

uint32_t **occupancy_mode** : 1

The value of Light LC Occupancy Mode state

uint32_t **light_onoff** : 1

The present value of Light LC Light OnOff state

uint32_t **target_light_onoff** : 1

The target value of Light LC Light OnOff state

uint32_t **occupancy** : 1

The value of Light LC Occupancy state

uint32_t **ambient_luxlevel** : 24

The value of Light LC Ambient LuxLevel state

uint16_t **linear_output**

1. Light LC Linear Output = $\max((\text{Lightness Out})^2/65535, \text{Regulator Output})$
2. If the Light LC Mode state is set to 0b1, the binding is enabled and upon a change of the Light LC Linear Output state, the following operation shall be performed: Light Lightness Linear = Light LC Linear Output
3. If the Light LC Mode state is set to 0b0, the binding is disabled (i.e., upon a change of the Light LC Linear Output state, no operation on the Light Lightness Linear state is performed). The value of Light LC Linear Output state

struct esp_ble_mesh_light_lc_property_state_t

Parameters of Light Property states. The Light LC Property states are read / write states that determine the configuration of a Light Lightness Controller. Each state is represented by a device property and is controlled by Light LC Property messages.

Public Members**uint32_t time_occupancy_delay**

A timing state that determines the delay for changing the Light LC Occupancy state upon receiving a Sensor Status message from an occupancy sensor. The value of Light LC Time Occupancy Delay state

uint32_t time_fade_on

A timing state that determines the time the controlled lights fade to the level determined by the Light LC Lightness On state. The value of Light LC Time Fade On state

uint32_t time_run_on

A timing state that determines the time the controlled lights stay at the level determined by the Light LC Lightness On state. The value of Light LC Time Run On state

uint32_t time_fade

A timing state that determines the time the controlled lights fade from the level determined by the Light LC Lightness On state to the level determined by the Light Lightness Prolong state. The value of Light LC Time Fade state

uint32_t time_prolong

A timing state that determines the time the controlled lights stay at the level determined by the Light LC Lightness Prolong state. The value of Light LC Time Prolong state

uint32_t time_fade_standby_auto

A timing state that determines the time the controlled lights fade from the level determined by the Light LC Lightness Prolong state to the level determined by the Light LC Lightness Standby state when the transition is automatic. The value of Light LC Time Fade Standby Auto state

uint32_t time_fade_standby_manual

A timing state that determines the time the controlled lights fade from the level determined by the Light LC Lightness Prolong state to the level determined by the Light LC Lightness Standby state when the transition is triggered by a change in the Light LC Light OnOff state. The value of Light LC Time Fade Standby Manual state

uint16_t lightness_on

A lightness state that determines the perceptible light lightness at the Occupancy and Run internal controller states. The value of Light LC Lightness On state

uint16_t lightness_prolong

A lightness state that determines the light lightness at the Prolong internal controller state. The value of Light LC Lightness Prolong state

uint16_t lightness_standby

A lightness state that determines the light lightness at the Standby internal controller state. The value of Light LC Lightness Standby state

uint16_t ambient_luxlevel_on

A uint16 state representing the Ambient LuxLevel level that determines if the controller transitions from the Light Control Standby state. The value of Light LC Ambient LuxLevel On state

uint16_t ambient_luxlevel_prolong

A uint16 state representing the required Ambient LuxLevel level in the Prolong state. The value of Light LC Ambient LuxLevel Prolong state

uint16_t ambient_luxlevel_standby

A uint16 state representing the required Ambient LuxLevel level in the Standby state. The value of Light LC Ambient LuxLevel Standby state

float regulator_kiu

A float32 state representing the integral coefficient that determines the integral part of the equation defining the output of the Light LC PI Feedback Regulator, when Light LC Ambient LuxLevel is less than LuxLevel Out. Valid range: 0.0 ~ 1000.0. The default value is 250.0. The value of Light LC Regulator Kiu state

float regulator_kid

A float32 state representing the integral coefficient that determines the integral part of the equation defining the output of the Light LC PI Feedback Regulator, when Light LC Ambient LuxLevel is greater than or equal to the value of the LuxLevel Out state. Valid range: 0.0 ~ 1000.0. The default value is 25.0. The value of Light LC Regulator Kid state

float regulator_kpu

A float32 state representing the proportional coefficient that determines the proportional part of the equation defining the output of the Light LC PI Feedback Regulator, when Light LC Ambient LuxLevel is less than the value of the LuxLevel Out state. Valid range: 0.0 ~ 1000.0. The default value is 80.0. The value of Light LC Regulator Kpu state

float regulator_kpd

A float32 state representing the proportional coefficient that determines the proportional part of the equation defining the output of the Light LC PI Feedback Regulator, when Light LC Ambient LuxLevel is greater than or equal to the value of the LuxLevel Out state. Valid range: 0.0 ~ 1000.0. The default value is 80.0. The value of Light LC Regulator Kpd state

int8_t regulator_accuracy

A int8 state representing the percentage accuracy of the Light LC PI Feedback Regulator. Valid range: 0.0 ~ 100.0. The default value is 2.0. The value of Light LC Regulator Accuracy state

uint32_t set_occupancy_to_1_delay

If the message Raw field contains a Raw Value for the Time Since Motion Sensed device property, which represents a value less than or equal to the value of the Light LC Occupancy Delay state, it shall delay setting the Light LC Occupancy state to Ob1 by the difference between the value of the Light LC Occupancy Delay state and the received Time Since Motion value. The value of the difference between value of the Light LC Occupancy Delay state and the received Time Since Motion value

struct esp_ble_mesh_light_lc_state_machine_t

Parameters of Light LC state machine

Public Members**uint8_t fade_on**

The value of transition time of Light LC Time Fade On

uint8_t fade

The value of transition time of Light LC Time Fade

uint8_t fade_standby_auto

The value of transition time of Light LC Time Fade Standby Auto

uint8_t fade_standby_manual

The value of transition time of Light LC Time Fade Standby Manual

struct esp_ble_mesh_light_lc_state_machine_t::[anonymous] trans_time

The Fade On, Fade, Fade Standby Auto, and Fade Standby Manual states are transition states that define the transition of the Lightness Out and LuxLevel Out states. This transition can be started as a result of the Light LC State Machine change or as a result of receiving the Light LC Light OnOff Set or Light LC Light Set Unacknowledged message. The value of transition time

esp_ble_mesh_light_lc_state_t state

The value of Light LC state machine state

struct k_delayed_work timer

Timer of Light LC state machine

struct esp_ble_mesh_light_control_t
Parameters of Light Lightness controller

Public Members

esp_ble_mesh_light_lc_state_t **state**
Parameters of Light LC state

esp_ble_mesh_light_lc_property_state_t **prop_state**
Parameters of Light LC Property state

esp_ble_mesh_light_lc_state_machine_t **state_machine**
Parameters of Light LC state machine

struct esp_ble_mesh_light_lc_srv_t
User data of Light LC Server Model

Public Members

esp_ble_mesh_model_t ***model**
Pointer to the Lighting LC Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**
Response control of the server model received messages

esp_ble_mesh_light_control_t ***lc**
Parameters of the Light controller

esp_ble_mesh_last_msg_info_t **last**
Parameters of the last received set message

esp_ble_mesh_state_transition_t **transition**
Parameters of state transition

struct esp_ble_mesh_light_lc_setup_srv_t
User data of Light LC Setup Server Model

Public Members

esp_ble_mesh_model_t ***model**
Pointer to the Lighting LC Setup Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t **rsp_ctrl**
Response control of the server model received messages

esp_ble_mesh_light_control_t ***lc**
Parameters of the Light controller

struct esp_ble_mesh_state_change_light_lightness_set_t
Parameter of Light Lightness Actual state change event

Public Members

uint16_t **lightness**
The value of Light Lightness Actual state

struct esp_ble_mesh_state_change_light_lightness_linear_set_t
Parameter of Light Lightness Linear state change event

Public Members**uint16_t lightness**

The value of Light Lightness Linear state

struct esp_ble_mesh_state_change_light_lightness_default_set_t

Parameter of Light Lightness Default state change event

Public Members**uint16_t lightness**

The value of Light Lightness Default state

struct esp_ble_mesh_state_change_light_lightness_range_set_t

Parameters of Light Lightness Range state change event

Public Members**uint16_t range_min**

The minimum value of Light Lightness Range state

uint16_t range_max

The maximum value of Light Lightness Range state

struct esp_ble_mesh_state_change_light_ctl_set_t

Parameters of Light CTL state change event

Public Members**uint16_t lightness**

The value of Light CTL Lightness state

uint16_t temperature

The value of Light CTL Temperature state

int16_t delta_uv

The value of Light CTL Delta UV state

struct esp_ble_mesh_state_change_light_ctl_temperature_set_t

Parameters of Light CTL Temperature state change event

Public Members**uint16_t temperature**

The value of Light CTL Temperature state

int16_t delta_uv

The value of Light CTL Delta UV state

struct esp_ble_mesh_state_change_light_ctl_temperature_range_set_t

Parameters of Light CTL Temperature Range state change event

Public Members**uint16_t range_min**

The minimum value of Light CTL Temperature Range state

uint16_t range_max

The maximum value of Light CTL Temperature Range state

struct esp_ble_mesh_state_change_light_ctl_default_set_t
Parameters of Light CTL Default state change event

Public Members

uint16_t lightness
The value of Light Lightness Default state

uint16_t temperature
The value of Light CTL Temperature Default state

int16_t delta_uv
The value of Light CTL Delta UV Default state

struct esp_ble_mesh_state_change_light_hsl_set_t
Parameters of Light HSL state change event

Public Members

uint16_t lightness
The value of Light HSL Lightness state

uint16_t hue
The value of Light HSL Hue state

uint16_t saturation
The value of Light HSL Saturation state

struct esp_ble_mesh_state_change_light_hsl_hue_set_t
Parameter of Light HSL Hue state change event

Public Members

uint16_t hue
The value of Light HSL Hue state

struct esp_ble_mesh_state_change_light_hsl_saturation_set_t
Parameter of Light HSL Saturation state change event

Public Members

uint16_t saturation
The value of Light HSL Saturation state

struct esp_ble_mesh_state_change_light_hsl_default_set_t
Parameters of Light HSL Default state change event

Public Members

uint16_t lightness
The value of Light HSL Lightness Default state

uint16_t hue
The value of Light HSL Hue Default state

uint16_t saturation
The value of Light HSL Saturation Default state

struct esp_ble_mesh_state_change_light_hsl_range_set_t
Parameters of Light HSL Range state change event

Public Members

uint16_t hue_range_min
The minimum hue value of Light HSL Range state

uint16_t hue_range_max
The maximum hue value of Light HSL Range state

uint16_t saturation_range_min
The minimum saturation value of Light HSL Range state

uint16_t saturation_range_max
The maximum saturation value of Light HSL Range state

struct esp_ble_mesh_state_change_light_xyl_set_t
Parameters of Light xyL state change event

Public Members

uint16_t lightness
The value of Light xyL Lightness state

uint16_t x
The value of Light xyL x state

uint16_t y
The value of Light xyL y state

struct esp_ble_mesh_state_change_light_xyl_default_set_t
Parameters of Light xyL Default state change event

Public Members

uint16_t lightness
The value of Light Lightness Default state

uint16_t x
The value of Light xyL x Default state

uint16_t y
The value of Light xyL y Default state

struct esp_ble_mesh_state_change_light_xyl_range_set_t
Parameters of Light xyL Range state change event

Public Members

uint16_t x_range_min
The minimum value of Light xyL x Range state

uint16_t x_range_max
The maximum value of Light xyL x Range state

uint16_t y_range_min
The minimum value of Light xyL y Range state

uint16_t y_range_max
The maximum value of Light xyL y Range state

struct esp_ble_mesh_state_change_light_lc_mode_set_t
Parameter of Light LC Mode state change event

Public Members**uint8_t mode**

The value of Light LC Mode state

struct esp_ble_mesh_state_change_light_lc_om_set_t

Parameter of Light LC Occupancy Mode state change event

Public Members**uint8_t mode**

The value of Light LC Occupancy Mode state

struct esp_ble_mesh_state_change_light_lc_light_onoff_set_t

Parameter of Light LC Light OnOff state change event

Public Members**uint8_t onoff**

The value of Light LC Light OnOff state

struct esp_ble_mesh_state_change_light_lc_property_set_t

Parameters of Light LC Property state change event

Public Members**uint16_t property_id**

The property id of Light LC Property state

struct net_buf_simple *property_value

The property value of Light LC Property state

struct esp_ble_mesh_state_change_sensor_status_t

Parameters of Sensor Status state change event

Public Members**uint16_t property_id**

The value of Sensor Property ID

uint8_t occupancy

The value of Light LC Occupancy state

uint32_t set_occupancy_to_1_delay

The value of Light LC Set Occupancy to 1 Delay state

uint32_t ambient_luxlevel

The value of Light LC Ambient Luxlevel state

union esp_ble_mesh_state_change_sensor_status_t::[anonymous] state

Parameters of Sensor Status related state

struct esp_ble_mesh_server_rcv_light_lc_property_get_t

Context of the received Light LC Property Get message

Public Members

`uint16_t property_id`
Property ID identifying a Light LC Property

struct esp_ble_mesh_server_rcv_light_lightness_set_t
Context of the received Light Lightness Set message

Public Members

`bool op_en`
Indicate if optional parameters are included

`uint16_t lightness`
Target value of light lightness actual state

`uint8_t tid`
Transaction ID

`uint8_t trans_time`
Time to complete state transition (optional)

`uint8_t delay`
Indicate message execution delay (C.1)

struct esp_ble_mesh_server_rcv_light_lightness_linear_set_t
Context of the received Light Lightness Linear Set message

Public Members

`bool op_en`
Indicate if optional parameters are included

`uint16_t lightness`
Target value of light lightness linear state

`uint8_t tid`
Transaction ID

`uint8_t trans_time`
Time to complete state transition (optional)

`uint8_t delay`
Indicate message execution delay (C.1)

struct esp_ble_mesh_server_rcv_light_lightness_default_set_t
Context of the received Light Lightness Default Set message

Public Members

`uint16_t lightness`
The value of the Light Lightness Default state

struct esp_ble_mesh_server_rcv_light_lightness_range_set_t
Context of the received Light Lightness Range Set message

Public Members

`uint16_t range_min`
Value of range min field of light lightness range state

uint16_t range_max
Value of range max field of light lightness range state

struct esp_ble_mesh_server_rcv_light_ctl_set_t
Context of the received Light CTL Set message

Public Members

bool op_en
Indicate if optional parameters are included

uint16_t lightness
Target value of light ctl lightness state

uint16_t temperature
Target value of light ctl temperature state

int16_t delta_uv
Target value of light ctl delta UV state

uint8_t tid
Transaction ID

uint8_t trans_time
Time to complete state transition (optional)

uint8_t delay
Indicate message execution delay (C.1)

struct esp_ble_mesh_server_rcv_light_ctl_temperature_set_t
Context of the received Light CTL Temperature Set message

Public Members

bool op_en
Indicate if optional parameters are included

uint16_t temperature
Target value of light ctl temperature state

int16_t delta_uv
Target value of light ctl delta UV state

uint8_t tid
Transaction ID

uint8_t trans_time
Time to complete state transition (optional)

uint8_t delay
Indicate message execution delay (C.1)

struct esp_ble_mesh_server_rcv_light_ctl_temperature_range_set_t
Context of the received Light CTL Temperature Range Set message

Public Members

uint16_t range_min
Value of temperature range min field of light ctl temperature range state

uint16_t range_max
Value of temperature range max field of light ctl temperature range state

struct esp_ble_mesh_server_rcv_light_ctl_default_set_t
Context of the received Light CTL Default Set message

Public Members

uint16_t lightness
Value of light lightness default state

uint16_t temperature
Value of light temperature default state

int16_t delta_uv
Value of light delta UV default state

struct esp_ble_mesh_server_rcv_light_hsl_set_t
Context of the received Light HSL Set message

Public Members

bool op_en
Indicate if optional parameters are included

uint16_t lightness
Target value of light hsl lightness state

uint16_t hue
Target value of light hsl hue state

uint16_t saturation
Target value of light hsl saturation state

uint8_t tid
Transaction ID

uint8_t trans_time
Time to complete state transition (optional)

uint8_t delay
Indicate message execution delay (C.1)

struct esp_ble_mesh_server_rcv_light_hsl_hue_set_t
Context of the received Light HSL Hue Set message

Public Members

bool op_en
Indicate if optional parameters are included

uint16_t hue
Target value of light hsl hue state

uint8_t tid
Transaction ID

uint8_t trans_time
Time to complete state transition (optional)

uint8_t delay
Indicate message execution delay (C.1)

struct esp_ble_mesh_server_rcv_light_hsl_saturation_set_t
Context of the received Light HSL Saturation Set message

Public Members

bool **op_en**
Indicate if optional parameters are included

uint16_t **saturation**
Target value of light hsl hue state

uint8_t **tid**
Transaction ID

uint8_t **trans_time**
Time to complete state transition (optional)

uint8_t **delay**
Indicate message execution delay (C.1)

struct esp_ble_mesh_server_rcv_light_hsl_default_set_t
Context of the received Light HSL Default Set message

Public Members

uint16_t **lightness**
Value of light lightness default state

uint16_t **hue**
Value of light hue default state

uint16_t **saturation**
Value of light saturation default state

struct esp_ble_mesh_server_rcv_light_hsl_range_set_t
Context of the received Light HSL Range Set message

Public Members

uint16_t **hue_range_min**
Value of hue range min field of light hsl hue range state

uint16_t **hue_range_max**
Value of hue range max field of light hsl hue range state

uint16_t **saturation_range_min**
Value of saturation range min field of light hsl saturation range state

uint16_t **saturation_range_max**
Value of saturation range max field of light hsl saturation range state

struct esp_ble_mesh_server_rcv_light_xyl_set_t
Context of the received Light xyL Set message

Public Members

bool **op_en**
Indicate whether optional parameters included

uint16_t **lightness**
The target value of the Light xyL Lightness state

uint16_t **x**
The target value of the Light xyL x state

uint16_t y
The target value of the Light xyL y state

uint8_t tid
Transaction Identifier

uint8_t trans_time
Time to complete state transition (optional)

uint8_t delay
Indicate message execution delay (C.1)

struct esp_ble_mesh_server_rcv_light_xyl_default_set_t
Context of the received Light xyL Default Set message

Public Members

uint16_t lightness
The value of the Light Lightness Default state

uint16_t x
The value of the Light xyL x Default state

uint16_t y
The value of the Light xyL y Default state

struct esp_ble_mesh_server_rcv_light_xyl_range_set_t
Context of the received Light xyL Range Set message

Public Members

uint16_t x_range_min
The value of the xyL x Range Min field of the Light xyL x Range state

uint16_t x_range_max
The value of the xyL x Range Max field of the Light xyL x Range state

uint16_t y_range_min
The value of the xyL y Range Min field of the Light xyL y Range state

uint16_t y_range_max
The value of the xyL y Range Max field of the Light xyL y Range state

struct esp_ble_mesh_server_rcv_light_lc_mode_set_t
Context of the received Light LC Mode Set message

Public Members

uint8_t mode
The target value of the Light LC Mode state

struct esp_ble_mesh_server_rcv_light_lc_om_set_t
Context of the received Light OM Set message

Public Members

uint8_t mode
The target value of the Light LC Occupancy Mode state

struct esp_ble_mesh_server_rcv_light_lc_light_onoff_set_t
Context of the received Light LC Light OnOff Set message

Public Members

bool op_en
Indicate whether optional parameters included

uint8_t light_onoff
The target value of the Light LC Light OnOff state

uint8_t tid
Transaction Identifier

uint8_t trans_time
Time to complete state transition (optional)

uint8_t delay
Indicate message execution delay (C.1)

struct esp_ble_mesh_server_rcv_light_lc_property_set_t
Context of the received Light LC Property Set message

Public Members

uint16_t property_id
Property ID identifying a Light LC Property

struct net_buf_simple *property_value
Raw value for the Light LC Property

struct esp_ble_mesh_server_rcv_sensor_status_t
Context of the received Sensor Status message

Public Members

struct net_buf_simple *data
Value of sensor data state (optional)

struct esp_ble_mesh_lighting_server_cb_param_t
Lighting Server Model callback parameters

Public Members

***esp_ble_mesh_model_t* *model**
Pointer to Lighting Server Models

***esp_ble_mesh_msg_ctx_t* ctx**
Context of the received messages

***esp_ble_mesh_lighting_server_cb_value_t* value**
Value of the received Lighting Messages

Macros

ESP_BLE_MESH_MODEL_LIGHT_LIGHTNESS_CLI (cli_pub, cli_data)
Define a new Light Lightness Client Model.

Note This API needs to be called for each element on which the application needs to have a Light Lightness Client Model.

Return New Light Lightness Client Model instance.

Parameters

- **cli_pub**: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- **cli_data**: Pointer to the unique struct *esp_ble_mesh_client_t*.

ESP_BLE_MESH_MODEL_LIGHT_CTL_CLI (cli_pub, cli_data)

Define a new Light CTL Client Model.

Note This API needs to be called for each element on which the application needs to have a Light CTL Client Model.

Return New Light CTL Client Model instance.

Parameters

- cli_pub: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- cli_data: Pointer to the unique struct *esp_ble_mesh_client_t*.

ESP_BLE_MESH_MODEL_LIGHT_HSL_CLI (cli_pub, cli_data)

Define a new Light HSL Client Model.

Note This API needs to be called for each element on which the application needs to have a Light HSL Client Model.

Return New Light HSL Client Model instance.

Parameters

- cli_pub: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- cli_data: Pointer to the unique struct *esp_ble_mesh_client_t*.

ESP_BLE_MESH_MODEL_LIGHT_XYL_CLI (cli_pub, cli_data)

Define a new Light xyL Client Model.

Note This API needs to be called for each element on which the application needs to have a Light xyL Client Model.

Return New Light xyL Client Model instance.

Parameters

- cli_pub: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- cli_data: Pointer to the unique struct *esp_ble_mesh_client_t*.

ESP_BLE_MESH_MODEL_LIGHT_LC_CLI (cli_pub, cli_data)

Define a new Light LC Client Model.

Note This API needs to be called for each element on which the application needs to have a Light LC Client Model.

Return New Light LC Client Model instance.

Parameters

- cli_pub: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- cli_data: Pointer to the unique struct *esp_ble_mesh_client_t*.

ESP_BLE_MESH_MODEL_LIGHT_LIGHTNESS_SRV (srv_pub, srv_data)

Lighting Server Models related context.

Define a new Light Lightness Server Model.

Note 1. The Light Lightness Server model extends the Generic Power OnOff Server model and the Generic Level Server model. When this model is present on an Element, the corresponding Light Lightness Setup Server model shall also be present.

1. This model shall support model publication and model subscription.

Return New Light Lightness Server Model instance.

Parameters

- srv_pub: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- srv_data: Pointer to the unique struct *esp_ble_mesh_light_lightness_srv_t*.

ESP_BLE_MESH_MODEL_LIGHT_LIGHTNESS_SETUP_SRV (srv_pub, srv_data)

Define a new Light Lightness Setup Server Model.

Note 1. The Light Lightness Setup Server model extends the Light Lightness Server model and the Generic Power OnOff Setup Server model.

1. This model shall support model subscription.

Return New Light Lightness Setup Server Model instance.

Parameters

- srv_pub: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- srv_data: Pointer to the unique struct *esp_ble_mesh_light_lightness_setup_srv_t*.

ESP_BLE_MESH_MODEL_LIGHT_CTL_SRV (srv_pub, srv_data)

Define a new Light CTL Server Model.

Note 1. The Light CTL Server model extends the Light Lightness Server model. When this model is present on an Element, the corresponding Light CTL Temperature Server model and the corresponding Light CTL Setup Server model shall also be present.

1. This model shall support model publication and model subscription.
2. The model requires two elements: the main element and the Temperature element. The Temperature element contains the corresponding Light CTL Temperature Server model and an instance of a Generic Level state bound to the Light CTL Temperature state on the Temperature element. The Light CTL Temperature state on the Temperature element is bound to the Light CTL state on the main element.

Return New Light CTL Server Model instance.

Parameters

- `srv_pub`: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- `srv_data`: Pointer to the unique struct *esp_ble_mesh_light_ctl_srv_t*.

ESP_BLE_MESH_MODEL_LIGHT_CTL_SETUP_SRV (srv_pub, srv_data)

Define a new Light CTL Setup Server Model.

Note 1. The Light CTL Setup Server model extends the Light CTL Server and the Light Lightness Setup Server.

1. This model shall support model subscription.

Return New Light CTL Setup Server Model instance.

Parameters

- `srv_pub`: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- `srv_data`: Pointer to the unique struct *esp_ble_mesh_light_ctl_setup_srv_t*.

ESP_BLE_MESH_MODEL_LIGHT_CTL_TEMP_SRV (srv_pub, srv_data)

Define a new Light CTL Temperature Server Model.

Note 1. The Light CTL Temperature Server model extends the Generic Level Server model.

1. This model shall support model publication and model subscription.

Return New Light CTL Temperature Server Model instance.

Parameters

- `srv_pub`: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- `srv_data`: Pointer to the unique struct *esp_ble_mesh_light_ctl_temp_srv_t*.

ESP_BLE_MESH_MODEL_LIGHT_HSL_SRV (srv_pub, srv_data)

Define a new Light HSL Server Model.

Note 1. The Light HSL Server model extends the Light Lightness Server model. When this model is present on an Element, the corresponding Light HSL Hue Server model and the corresponding Light HSL Saturation Server model and the corresponding Light HSL Setup Server model shall also be present.

1. This model shall support model publication and model subscription.
2. The model requires three elements: the main element and the Hue element and the Saturation element. The Hue element contains the corresponding Light HSL Hue Server model and an instance of a Generic Level state bound to the Light HSL Hue state on the Hue element. The Saturation element contains the corresponding Light HSL Saturation Server model and an instance of a Generic Level state bound to the Light HSL Saturation state on the Saturation element. The Light HSL Hue state on the Hue element is bound to the Light HSL state on the main element and the Light HSL Saturation state on the Saturation element is bound to the Light HSL state on the main element.

Return New Light HSL Server Model instance.

Parameters

- `srv_pub`: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- `srv_data`: Pointer to the unique struct *esp_ble_mesh_light_hsl_srv_t*.

ESP_BLE_MESH_MODEL_LIGHT_HSL_SETUP_SRV (srv_pub, srv_data)

Define a new Light HSL Setup Server Model.

Note 1. The Light HSL Setup Server model extends the Light HSL Server and the Light Lightness Setup Server.

1. This model shall support model subscription.

Return New Light HSL Setup Server Model instance.

Parameters

- `srv_pub`: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- `srv_data`: Pointer to the unique struct *esp_ble_mesh_light_hsl_setup_srv_t*.

ESP_BLE_MESH_MODEL_LIGHT_HSL_HUE_SRV (`srv_pub`, `srv_data`)

Define a new Light HSL Hue Server Model.

Note 1. The Light HSL Hue Server model extends the Generic Level Server model. This model is associated with the Light HSL Server model.

1. This model shall support model publication and model subscription.

Return New Light HSL Hue Server Model instance.

Parameters

- `srv_pub`: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- `srv_data`: Pointer to the unique struct *esp_ble_mesh_light_hsl_hue_srv_t*.

ESP_BLE_MESH_MODEL_LIGHT_HSL_SAT_SRV (`srv_pub`, `srv_data`)

Define a new Light HSL Saturation Server Model.

Note 1. The Light HSL Saturation Server model extends the Generic Level Server model. This model is associated with the Light HSL Server model.

1. This model shall support model publication and model subscription.

Return New Light HSL Saturation Server Model instance.

Parameters

- `srv_pub`: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- `srv_data`: Pointer to the unique struct *esp_ble_mesh_light_hsl_sat_srv_t*.

ESP_BLE_MESH_MODEL_LIGHT_XYL_SRV (`srv_pub`, `srv_data`)

Define a new Light xyL Server Model.

Note 1. The Light xyL Server model extends the Light Lightness Server model. When this model is present on an Element, the corresponding Light xyL Setup Server model shall also be present.

1. This model shall support model publication and model subscription.

Return New Light xyL Server Model instance.

Parameters

- `srv_pub`: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- `srv_data`: Pointer to the unique struct *esp_ble_mesh_light_xyl_srv_t*.

ESP_BLE_MESH_MODEL_LIGHT_XYL_SETUP_SRV (`srv_pub`, `srv_data`)

Define a new Light xyL Setup Server Model.

Note 1. The Light xyL Setup Server model extends the Light xyL Server and the Light Lightness Setup Server.

1. This model shall support model subscription.

Return New Light xyL Setup Server Model instance.

Parameters

- `srv_pub`: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- `srv_data`: Pointer to the unique struct *esp_ble_mesh_light_xyl_setup_srv_t*.

ESP_BLE_MESH_MODEL_LIGHT_LC_SRV (`srv_pub`, `srv_data`)

Define a new Light LC Server Model.

Note 1. The Light LC (Lightness Control) Server model extends the Light Lightness Server model and the Generic OnOff Server model. When this model is present on an Element, the corresponding Light LC Setup Server model shall also be present.

1. This model shall support model publication and model subscription.
2. This model may be used to represent an element that is a client to a Sensor Server model and controls the Light Lightness Actual state via defined state bindings.

Return New Light LC Server Model instance.

Parameters

- `srv_pub`: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- `srv_data`: Pointer to the unique struct *esp_ble_mesh_light_lc_srv_t*.

ESP_BLE_MESH_MODEL_LIGHT_LC_SETUP_SRV (`srv_pub`, `srv_data`)

Define a new Light LC Setup Server Model.

- Note**
1. The Light LC (Lightness Control) Setup model extends the Light LC Server model.
 1. This model shall support model publication and model subscription.
 2. This model may be used to configure setup parameters for the Light LC Server model.

Return New Light LC Setup Server Model instance.

Parameters

- `srv_pub`: Pointer to the unique struct `esp_ble_mesh_model_pub_t`.
- `srv_data`: Pointer to the unique struct `esp_ble_mesh_light_lc_setup_srv_t`.

Type Definitions

```
typedef void (*esp_ble_mesh_light_client_cb_t) (esp_ble_mesh_light_client_cb_event_t
                                              event, esp_ble_mesh_light_client_cb_param_t
                                              *param)
```

Bluetooth Mesh Light Client Model function.

Lighting Client Model callback function type

Parameters

- `event`: Event type
- `param`: Pointer to callback parameter

```
typedef void (*esp_ble_mesh_lighting_server_cb_t) (esp_ble_mesh_lighting_server_cb_event_t
                                                  event,
                                                  esp_ble_mesh_lighting_server_cb_param_t
                                                  *param)
```

Bluetooth Mesh Lighting Server Model function.

Lighting Server Model callback function type

Parameters

- `event`: Event type
- `param`: Pointer to callback parameter

Enumerations

```
enum esp_ble_mesh_light_client_cb_event_t
This enum value is the event of Lighting Client Model
```

Values:

```
ESP_BLE_MESH_LIGHT_CLIENT_GET_STATE_EVT
ESP_BLE_MESH_LIGHT_CLIENT_SET_STATE_EVT
ESP_BLE_MESH_LIGHT_CLIENT_PUBLISH_EVT
ESP_BLE_MESH_LIGHT_CLIENT_TIMEOUT_EVT
ESP_BLE_MESH_LIGHT_CLIENT_EVT_MAX
```

```
enum esp_ble_mesh_lc_state_t
This enum value is the Light LC State Machine states
```

Values:

```
ESP_BLE_MESH_LC_OFF
ESP_BLE_MESH_LC_STANDBY
ESP_BLE_MESH_LC_FADE_ON
ESP_BLE_MESH_LC_RUN
ESP_BLE_MESH_LC_FADE
ESP_BLE_MESH_LC_PROLONG
ESP_BLE_MESH_LC_FADE_STANDBY_AUTO
ESP_BLE_MESH_LC_FADE_STANDBY_MANUAL
```


enum esp_ble_mesh_lighting_server_cb_event_t

This enum value is the event of Lighting Server Model

Values:

ESP_BLE_MESH_LIGHTING_SERVER_STATE_CHANGE_EVT

1. When `get_auto_rsp` is set to `ESP_BLE_MESH_SERVER_AUTO_RSP`, no event will be callback to the application layer when Lighting Get messages are received.
2. When `set_auto_rsp` is set to `ESP_BLE_MESH_SERVER_AUTO_RSP`, this event will be callback to the application layer when Lighting Set/Set Unack messages are received.

ESP_BLE_MESH_LIGHTING_SERVER_RECV_GET_MSG_EVT

When `get_auto_rsp` is set to `ESP_BLE_MESH_SERVER_RSP_BY_APP`, this event will be callback to the application layer when Lighting Get messages are received.

ESP_BLE_MESH_LIGHTING_SERVER_RECV_SET_MSG_EVT

When `set_auto_rsp` is set to `ESP_BLE_MESH_SERVER_RSP_BY_APP`, this event will be callback to the application layer when Lighting Set/Set Unack messages are received.

ESP_BLE_MESH_LIGHTING_SERVER_RECV_STATUS_MSG_EVT

When `status_auto_rsp` is set to `ESP_BLE_MESH_SERVER_RSP_BY_APP`, this event will be callback to the application layer when Sensor Status message is received.

ESP_BLE_MESH_LIGHTING_SERVER_EVT_MAX**Time and Scenes Client/Server Models****Header File**

- [bt/esp_ble_mesh/api/models/include/esp_ble_mesh_sensor_model_api.h](#)

Functions

esp_err_t **esp_ble_mesh_register_sensor_client_callback** (*esp_ble_mesh_sensor_client_cb_t* callback)

Register BLE Mesh Sensor Client Model callback.

Return `ESP_OK` on success or error code otherwise.

Parameters

- [in] `callback`: Pointer to the callback function.

esp_err_t **esp_ble_mesh_sensor_client_get_state** (*esp_ble_mesh_client_common_param_t* *params, *esp_ble_mesh_sensor_client_get_state_t* *get_state)

Get the value of Sensor Server Model states using the Sensor Client Model get messages.

Note If you want to know the opcodes and corresponding meanings accepted by this API, please refer to `esp_ble_mesh_sensor_message_opcode_t` in `esp_ble_mesh_defs.h`

Return `ESP_OK` on success or error code otherwise.

Parameters

- [in] `params`: Pointer to BLE Mesh common client parameters.
- [in] `get_state`: Pointer to sensor get message value. Shall not be set to NULL.

esp_err_t **esp_ble_mesh_sensor_client_set_state** (*esp_ble_mesh_client_common_param_t* *params, *esp_ble_mesh_sensor_client_set_state_t* *set_state)

Set the value of Sensor Server Model states using the Sensor Client Model set messages.

Note If you want to know the opcodes and corresponding meanings accepted by this API, please refer to `esp_ble_mesh_sensor_message_opcode_t` in `esp_ble_mesh_defs.h`

Return `ESP_OK` on success or error code otherwise.

Parameters

- [in] `params`: Pointer to BLE Mesh common client parameters.
- [in] `set_state`: Pointer to sensor set message value. Shall not be set to NULL.

esp_err_t **esp_ble_mesh_register_sensor_server_callback** (*esp_ble_mesh_sensor_server_cb_t* callback)

Register BLE Mesh Sensor Server Model callback.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] callback: Pointer to the callback function.

Unions

union esp_ble_mesh_sensor_client_get_state_t
#include <esp_ble_mesh_sensor_model_api.h> Sensor Client Model get message union.

Public Members

esp_ble_mesh_sensor_descriptor_get_t **descriptor_get**
For ESP_BLE_MESH_MODEL_OP_SENSOR_DESCRIPTOR_GET

esp_ble_mesh_sensor_cadence_get_t **cadence_get**
For ESP_BLE_MESH_MODEL_OP_SENSOR_CADENCE_GET

esp_ble_mesh_sensor_settings_get_t **settings_get**
For ESP_BLE_MESH_MODEL_OP_SENSOR_SETTINGS_GET

esp_ble_mesh_sensor_setting_get_t **setting_get**
For ESP_BLE_MESH_MODEL_OP_SENSOR_SETTING_GET

esp_ble_mesh_sensor_get_t **sensor_get**
For ESP_BLE_MESH_MODEL_OP_SENSOR_GET

esp_ble_mesh_sensor_column_get_t **column_get**
For ESP_BLE_MESH_MODEL_OP_SENSOR_COLUMN_GET

esp_ble_mesh_sensor_series_get_t **series_get**
For ESP_BLE_MESH_MODEL_OP_SENSOR_SERIES_GET

union esp_ble_mesh_sensor_client_set_state_t
#include <esp_ble_mesh_sensor_model_api.h> Sensor Client Model set message union.

Public Members

esp_ble_mesh_sensor_cadence_set_t **cadence_set** &
For ESP_BLE_MESH_MODEL_OP_SENSOR_CADENCE_SET
ESP_BLE_MESH_MODEL_OP_SENSOR_CADENCE_SET_UNACK

esp_ble_mesh_sensor_setting_set_t **setting_set** &
For ESP_BLE_MESH_MODEL_OP_SENSOR_SETTING_SET
ESP_BLE_MESH_MODEL_OP_SENSOR_SETTING_SET_UNACK

union esp_ble_mesh_sensor_client_status_cb_t
#include <esp_ble_mesh_sensor_model_api.h> Sensor Client Model received message union.

Public Members

esp_ble_mesh_sensor_descriptor_status_cb_t **descriptor_status**
For ESP_BLE_MESH_MODEL_OP_SENSOR_DESCRIPTOR_STATUS

esp_ble_mesh_sensor_cadence_status_cb_t **cadence_status**
For ESP_BLE_MESH_MODEL_OP_SENSOR_CADENCE_STATUS

esp_ble_mesh_sensor_settings_status_cb_t **settings_status**
For ESP_BLE_MESH_MODEL_OP_SENSOR_SETTINGS_STATUS

esp_ble_mesh_sensor_setting_status_cb_t **setting_status**
For ESP_BLE_MESH_MODEL_OP_SENSOR_SETTING_STATUS

esp_ble_mesh_sensor_status_cb_t **sensor_status**
For ESP_BLE_MESH_MODEL_OP_SENSOR_STATUS

esp_ble_mesh_sensor_column_status_cb_t **column_status**
For ESP_BLE_MESH_MODEL_OP_SENSOR_COLUMN_STATUS

esp_ble_mesh_sensor_series_status_cb_t **series_status**
For ESP_BLE_MESH_MODEL_OP_SENSOR_SERIES_STATUS

union esp_ble_mesh_sensor_server_state_change_t
#include <esp_ble_mesh_sensor_model_api.h> Sensor Server Model state change value union.

Public Members

esp_ble_mesh_state_change_sensor_cadence_set_t **sensor_cadence_set**
The rcv_op in ctx can be used to decide which state is changed.Sensor Cadence Set

esp_ble_mesh_state_change_sensor_setting_set_t **sensor_setting_set**
Sensor Setting Set

union esp_ble_mesh_sensor_server_rcv_get_msg_t
#include <esp_ble_mesh_sensor_model_api.h> Sensor Server Model received get message union.

Public Members

esp_ble_mesh_server_rcv_sensor_descriptor_get_t **sensor_descriptor**
Sensor Descriptor Get

esp_ble_mesh_server_rcv_sensor_cadence_get_t **sensor_cadence**
Sensor Cadence Get

esp_ble_mesh_server_rcv_sensor_settings_get_t **sensor_settings**
Sensor Settings Get

esp_ble_mesh_server_rcv_sensor_setting_get_t **sensor_setting**
Sensor Setting Get

esp_ble_mesh_server_rcv_sensor_get_t **sensor_data**
Sensor Get

esp_ble_mesh_server_rcv_sensor_column_get_t **sensor_column**
Sensor Column Get

esp_ble_mesh_server_rcv_sensor_series_get_t **sensor_series**
Sensor Series Get

union esp_ble_mesh_sensor_server_rcv_set_msg_t
#include <esp_ble_mesh_sensor_model_api.h> Sensor Server Model received set message union.

Public Members

esp_ble_mesh_server_rcv_sensor_cadence_set_t **sensor_cadence**
Sensor Cadence Set

esp_ble_mesh_server_rcv_sensor_setting_set_t **sensor_setting**
Sensor Setting Set

union esp_ble_mesh_sensor_server_cb_value_t
#include <esp_ble_mesh_sensor_model_api.h> Sensor Server Model callback value union.

Public Members

esp_ble_mesh_sensor_server_state_change_t **state_change**
ESP_BLE_MESH_SENSOR_SERVER_STATE_CHANGE_EVT

esp_ble_mesh_sensor_server_rcv_get_msg_t **get**
ESP_BLE_MESH_SENSOR_SERVER_RECV_GET_MSG_EVT

esp_ble_mesh_sensor_server_rcv_set_msg_t **set**
ESP_BLE_MESH_SENSOR_SERVER_RECV_SET_MSG_EVT

Structures

struct esp_ble_mesh_sensor_descriptor_get_t
Bluetooth Mesh Sensor Client Model Get and Set parameters structure.
Parameters of Sensor Descriptor Get

Public Members

bool **op_en**
Indicate if optional parameters are included

uint16_t **property_id**
Property ID of a sensor (optional)

struct esp_ble_mesh_sensor_cadence_get_t
Parameter of Sensor Cadence Get

Public Members

uint16_t **property_id**
Property ID of a sensor

struct esp_ble_mesh_sensor_cadence_set_t
Parameters of Sensor Cadence Set

Public Members

uint16_t **property_id**
Property ID for the sensor

uint8_t **fast_cadence_period_divisor** : 7
Divisor for the publish period

uint8_t **status_trigger_type** : 1
The unit and format of the Status Trigger Delta fields

struct net_buf_simple *status_trigger_delta_down
Delta down value that triggers a status message

struct net_buf_simple *status_trigger_delta_up
Delta up value that triggers a status message

uint8_t **status_min_interval**
Minimum interval between two consecutive Status messages

struct net_buf_simple *fast_cadence_low
Low value for the fast cadence range

struct net_buf_simple *fast_cadence_high
Fast value for the fast cadence range

struct esp_ble_mesh_sensor_settings_get_t
Parameter of Sensor Settings Get

Public Members

uint16_t **sensor_property_id**
Property ID of a sensor

struct esp_ble_mesh_sensor_setting_get_t
Parameters of Sensor Setting Get

Public Members

uint16_t **sensor_property_id**
Property ID of a sensor

uint16_t **sensor_setting_property_id**
Setting ID identifying a setting within a sensor

struct esp_ble_mesh_sensor_setting_set_t
Parameters of Sensor Setting Set

Public Members

uint16_t **sensor_property_id**
Property ID identifying a sensor

uint16_t **sensor_setting_property_id**
Setting ID identifying a setting within a sensor

struct net_buf_simple *sensor_setting_raw
Raw value for the setting

struct esp_ble_mesh_sensor_get_t
Parameters of Sensor Get

Public Members

bool **op_en**
Indicate if optional parameters are included

uint16_t **property_id**
Property ID for the sensor (optional)

struct esp_ble_mesh_sensor_column_get_t
Parameters of Sensor Column Get

Public Members

uint16_t **property_id**
Property identifying a sensor

struct net_buf_simple *raw_value_x
Raw value identifying a column

struct esp_ble_mesh_sensor_series_get_t
Parameters of Sensor Series Get

Public Members

bool **op_en**
Indicate if optional parameters are included

uint16_t **property_id**
Property identifying a sensor

struct net_buf_simple ***raw_value_x1**
Raw value identifying a starting column (optional)

struct net_buf_simple ***raw_value_x2**
Raw value identifying an ending column (C.1)

struct **esp_ble_mesh_sensor_descriptor_status_cb_t**
Bluetooth Mesh Sensor Client Model Get and Set callback parameters structure.

Parameter of Sensor Descriptor Status

Public Members

struct net_buf_simple ***descriptor**
Sequence of 8-octet sensor descriptors (optional)

struct **esp_ble_mesh_sensor_cadence_status_cb_t**
Parameters of Sensor Cadence Status

Public Members

uint16_t **property_id**
Property for the sensor

struct net_buf_simple ***sensor_cadence_value**
Value of sensor cadence state

struct **esp_ble_mesh_sensor_settings_status_cb_t**
Parameters of Sensor Settings Status

Public Members

uint16_t **sensor_property_id**
Property ID identifying a sensor

struct net_buf_simple ***sensor_setting_property_ids**
A sequence of N sensor setting property IDs (optional)

struct **esp_ble_mesh_sensor_setting_status_cb_t**
Parameters of Sensor Setting Status

Public Members

bool **op_en**
Indicate id optional parameters are included

uint16_t **sensor_property_id**
Property ID identifying a sensor

uint16_t **sensor_setting_property_id**
Setting ID identifying a setting within a sensor

`uint8_t sensor_setting_access`
Read/Write access rights for the setting (optional)

`struct net_buf_simple *sensor_setting_raw`
Raw value for the setting

`struct esp_ble_mesh_sensor_status_cb_t`
Parameter of Sensor Status

Public Members

`struct net_buf_simple *marshalled_sensor_data`
Value of sensor data state (optional)

`struct esp_ble_mesh_sensor_column_status_cb_t`
Parameters of Sensor Column Status

Public Members

`uint16_t property_id`
Property identifying a sensor and the Y axis

`struct net_buf_simple *sensor_column_value`
Left values of sensor column status

`struct esp_ble_mesh_sensor_series_status_cb_t`
Parameters of Sensor Series Status

Public Members

`uint16_t property_id`
Property identifying a sensor and the Y axis

`struct net_buf_simple *sensor_series_value`
Left values of sensor series status

`struct esp_ble_mesh_sensor_client_cb_param_t`
Sensor Client Model callback parameters

Public Members

`int error_code`
0: success, otherwise failure. For the error code values please refer to `errno.h` file. A negative sign is added to the standard error codes in `errno.h`.

`esp_ble_mesh_client_common_param_t *params`
The client common parameters.

`esp_ble_mesh_sensor_client_status_cb_t status_cb`
The sensor status message callback values

`struct esp_ble_mesh_sensor_descriptor_t`
Parameters of Sensor Descriptor state

Public Members

`uint32_t positive_tolerance : 12`
The value of Sensor Positive Tolerance field

uint32_t negative_tolerance : 12
The value of Sensor Negative Tolerance field

uint32_t sampling_function : 8
The value of Sensor Sampling Function field

uint8_t measure_period
The value of Sensor Measurement Period field

uint8_t update_interval
The value of Sensor Update Interval field

struct esp_ble_mesh_sensor_setting_t
Parameters of Sensor Setting state

Public Members

uint16_t property_id
The value of Sensor Setting Property ID field

uint8_t access
The value of Sensor Setting Access field

struct net_buf_simple *raw
The value of Sensor Setting Raw field

struct esp_ble_mesh_sensor_cadence_t
Parameters of Sensor Cadence state

Public Members

uint8_t period_divisor : 7
The value of Fast Cadence Period Divisor field

uint8_t trigger_type : 1
The value of Status Trigger Type field

struct net_buf_simple *trigger_delta_down
Note: The parameter “size” in `trigger_delta_down`, `trigger_delta_up`, `fast_cadence_low` & `fast_cadence_high` indicates the exact length of these four parameters, and they are associated with the Sensor Property ID. Users need to initialize the “size” precisely. The value of Status Trigger Delta Down field

struct net_buf_simple *trigger_delta_up
The value of Status Trigger Delta Up field

uint8_t min_interval
The value of Status Min Interval field

struct net_buf_simple *fast_cadence_low
The value of Fast Cadence Low field

struct net_buf_simple *fast_cadence_high
The value of Fast Cadence High field

struct esp_ble_mesh_sensor_data_t
Parameters of Sensor Data state

Public Members

uint8_t format : 1
Format A: The Length field is a 1-based uint4 value (valid range 0x0–0xF, representing range of 1–16).

Format B: The Length field is a 1-based uint7 value (valid range 0x0–0x7F, representing range of 1 – 127). The value 0x7F represents a length of zero. The value of the Sensor Data format

uint8_t length : 7
The value of the Sensor Data length

struct net_buf_simple *raw_value
The value of Sensor Data raw value

struct esp_ble_mesh_sensor_series_column_t
Parameters of Sensor Series Column state

Public Members

struct net_buf_simple *raw_value_x
The value of Sensor Raw Value X field

struct net_buf_simple *column_width
The value of Sensor Column Width field

struct net_buf_simple *raw_value_y
The value of Sensor Raw Value Y field

struct esp_ble_mesh_sensor_state_t
Parameters of Sensor states

Public Members

uint16_t sensor_property_id
The value of Sensor Property ID field

esp_ble_mesh_sensor_descriptor_t descriptor
Parameters of the Sensor Descriptor state

const uint8_t setting_count
Multiple Sensor Setting states may be present for each sensor. The Sensor Setting Property ID values shall be unique for each Sensor Property ID that identifies a sensor within an element.

esp_ble_mesh_sensor_setting_t *settings
Parameters of the Sensor Setting state

esp_ble_mesh_sensor_cadence_t *cadence
The Sensor Cadence state may be not supported by sensors based on device properties referencing “non-scalar characteristics” such as “histograms” or “composite characteristics”. Parameters of the Sensor Cadence state

esp_ble_mesh_sensor_data_t sensor_data
Parameters of the Sensor Data state

esp_ble_mesh_sensor_series_column_t series_column
Parameters of the Sensor Series Column state

struct esp_ble_mesh_sensor_srv_t
User data of Sensor Server Model

Public Members

esp_ble_mesh_model_t *model
Pointer to the Sensor Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t rsp_ctrl
Response control of the server model received messages

const uint8_t state_count
Sensor state count

esp_ble_mesh_sensor_state_t *states
Parameters of the Sensor states

struct esp_ble_mesh_sensor_setup_srv_t
User data of Sensor Setup Server Model

Public Members

esp_ble_mesh_model_t *model
Pointer to the Sensor Setup Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t rsp_ctrl
Response control of the server model received messages

const uint8_t state_count
Sensor state count

esp_ble_mesh_sensor_state_t *states
Parameters of the Sensor states

struct esp_ble_mesh_state_change_sensor_cadence_set_t
Parameters of Sensor Cadence Set state change event

Public Members

uint16_t **property_id**
The value of Sensor Property ID state

uint8_t **period_divisor** : 7
The value of Fast Cadence Period Divisor state

uint8_t **trigger_type** : 1
The value of Status Trigger Type state

struct net_buf_simple ***trigger_delta_down**
The value of Status Trigger Delta Down state

struct net_buf_simple ***trigger_delta_up**
The value of Status Trigger Delta Up state

uint8_t **min_interval**
The value of Status Min Interval state

struct net_buf_simple ***fast_cadence_low**
The value of Fast Cadence Low state

struct net_buf_simple ***fast_cadence_high**
The value of Fast Cadence High state

struct esp_ble_mesh_state_change_sensor_setting_set_t
Parameters of Sensor Setting Set state change event

Public Members

uint16_t **property_id**
The value of Sensor Property ID state

uint16_t **setting_property_id**
The value of Sensor Setting Property ID state

struct net_buf_simple ***setting_value**

The value of Sensor Property Value state

struct esp_ble_mesh_server_rcv_sensor_descriptor_get_t

Context of the received Sensor Descriptor Get message

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **property_id**

Property ID of a sensor (optional)

struct esp_ble_mesh_server_rcv_sensor_cadence_get_t

Context of the received Sensor Cadence Get message

Public Members

uint16_t **property_id**

Property ID of a sensor

struct esp_ble_mesh_server_rcv_sensor_settings_get_t

Context of the received Sensor Settings Get message

Public Members

uint16_t **property_id**

Property ID of a sensor

struct esp_ble_mesh_server_rcv_sensor_setting_get_t

Context of the received Sensor Setting Get message

Public Members

uint16_t **property_id**

Property ID of a sensor

uint16_t **setting_property_id**

Setting ID identifying a setting within a sensor

struct esp_ble_mesh_server_rcv_sensor_get_t

Context of the received Sensor Get message

Public Members

bool **op_en**

Indicate if optional parameters are included

uint16_t **property_id**

Property ID for the sensor (optional)

struct esp_ble_mesh_server_rcv_sensor_column_get_t

Context of the received Sensor Column Get message

Public Members**uint16_t property_id**

Property identifying a sensor

struct net_buf_simple *raw_value_x

Raw value identifying a column

struct esp_ble_mesh_server_rcv_sensor_series_get_t

Context of the received Sensor Series Get message

Public Members**bool op_en**

Indicate if optional parameters are included

uint16_t property_id

Property identifying a sensor

struct net_buf_simple *raw_value

Raw value containing X1 and X2 (optional)

struct esp_ble_mesh_server_rcv_sensor_cadence_set_t

Context of the received Sensor Cadence Set message

Public Members**uint16_t property_id**

Property ID for the sensor

struct net_buf_simple *cadence

Value of Sensor Cadence state

struct esp_ble_mesh_server_rcv_sensor_setting_set_t

Context of the received Sensor Setting Set message

Public Members**uint16_t property_id**

Property ID identifying a sensor

uint16_t setting_property_id

Setting ID identifying a setting within a sensor

struct net_buf_simple *setting_raw

Raw value for the setting

struct esp_ble_mesh_sensor_server_cb_param_t

Sensor Server Model callback parameters

Public Members***esp_ble_mesh_model_t* *model**

Pointer to Sensor Server Models

***esp_ble_mesh_msg_ctx_t* ctx**

Context of the received messages

***esp_ble_mesh_sensor_server_cb_value_t* value**

Value of the received Sensor Messages

Macros**ESP_BLE_MESH_MODEL_SENSOR_CLI** (cli_pub, cli_data)

Define a new Sensor Client Model.

Note This API needs to be called for each element on which the application needs to have a Sensor Client Model.**Return** New Sensor Client Model instance.**Parameters**

- cli_pub: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- cli_data: Pointer to the unique struct *esp_ble_mesh_client_t*.

ESP_BLE_MESH_MODEL_SENSOR_SRV (srv_pub, srv_data)

Sensor Server Models related context.

Define a new Sensor Server Model.

Note 1. The Sensor Server model is a root model. When this model is present on an element, the corresponding Sensor Setup Server model shall also be present.

1. This model shall support model publication and model subscription.

Return New Sensor Server Model instance.**Parameters**

- srv_pub: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- srv_data: Pointer to the unique struct *esp_ble_mesh_sensor_srv_t*.

ESP_BLE_MESH_MODEL_SENSOR_SETUP_SRV (srv_pub, srv_data)

Define a new Sensor Setup Server Model.

Note 1. The Sensor Setup Server model extends the Sensor Server model.

1. This model shall support model publication and model subscription.

Return New Sensor Setup Server Model instance.**Parameters**

- srv_pub: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- srv_data: Pointer to the unique struct *esp_ble_mesh_sensor_setup_srv_t*.

ESP_BLE_MESH_INVALID_SENSOR_PROPERTY_ID

Invalid Sensor Property ID

ESP_BLE_MESH_SENSOR_PROPERTY_ID_LEN

Length of Sensor Property ID

ESP_BLE_MESH_SENSOR_DESCRIPTOR_LEN

Length of Sensor Descriptor state

ESP_BLE_MESH_SENSOR_UNSPECIFIED_POS_TOLERANCE

Unspecified Sensor Positive Tolerance

ESP_BLE_MESH_SENSOR_UNSPECIFIED_NEG_TOLERANCE

Unspecified Sensor Negative Tolerance

ESP_BLE_MESH_SENSOR_NOT_APPL_MEASURE_PERIOD

Not applicable Sensor Measurement Period

ESP_BLE_MESH_SENSOR_NOT_APPL_UPDATE_INTERVAL

Not applicable Sensor Update Interval

ESP_BLE_MESH_INVALID_SENSOR_SETTING_PROPERTY_ID

Invalid Sensor Setting Property ID

ESP_BLE_MESH_SENSOR_SETTING_PROPERTY_ID_LEN

Length of Sensor Setting Property ID

ESP_BLE_MESH_SENSOR_SETTING_ACCESS_LEN

Length of Sensor Setting Access

ESP_BLE_MESH_SENSOR_SETTING_ACCESS_READ

Sensor Setting Access - Read

ESP_BLE_MESH_SENSOR_SETTING_ACCESS_READ_WRITE

Sensor Setting Access - Read & Write

ESP_BLE_MESH_SENSOR_DIVISOR_TRIGGER_TYPE_LEN

Length of Sensor Divisor Trigger Type

ESP_BLE_MESH_SENSOR_STATUS_MIN_INTERVAL_LEN

Length of Sensor Status Min Interval

ESP_BLE_MESH_SENSOR_PERIOD_DIVISOR_MAX_VALUE

Maximum value of Sensor Period Divisor

ESP_BLE_MESH_SENSOR_STATUS_MIN_INTERVAL_MAX

Maximum value of Sensor Status Min Interval

ESP_BLE_MESH_SENSOR_STATUS_TRIGGER_TYPE_CHAR

Sensor Status Trigger Type - Format Type of the characteristic that the Sensor Property ID state references

ESP_BLE_MESH_SENSOR_STATUS_TRIGGER_TYPE_UINT16

Sensor Status Trigger Type - Format Type “uint16”

ESP_BLE_MESH_SENSOR_DATA_FORMAT_A

Sensor Data Format A

ESP_BLE_MESH_SENSOR_DATA_FORMAT_B

Sensor Data Format B

ESP_BLE_MESH_SENSOR_DATA_FORMAT_A_MP_ID_LEN

MPID length of Sensor Data Format A

ESP_BLE_MESH_SENSOR_DATA_FORMAT_B_MP_ID_LEN

MPID length of Sensor Data Format B

ESP_BLE_MESH_SENSOR_DATA_ZERO_LEN

Zero length of Sensor Data.

Note: The Length field is a 1-based uint7 value (valid range 0x0–0x7F, representing range of 1–127). The value 0x7F represents a length of zero.

ESP_BLE_MESH_GET_SENSOR_DATA_FORMAT (_data)

Get format of the sensor data.

Note Multiple sensor data may be concatenated. Make sure the _data pointer is updated before getting the format of the corresponding sensor data.

Return Format of the sensor data.

Parameters

- _data: Pointer to the start of the sensor data.

ESP_BLE_MESH_GET_SENSOR_DATA_LENGTH (_data, _fmt)

Get length of the sensor data.

Note Multiple sensor data may be concatenated. Make sure the _data pointer is updated before getting the length of the corresponding sensor data.

Return Length (zero-based) of the sensor data.

Parameters

- _data: Pointer to the start of the sensor data.
- _fmt: Format of the sensor data.

ESP_BLE_MESH_GET_SENSOR_DATA_PROPERTY_ID (_data, _fmt)

Get Sensor Property ID of the sensor data.

Note Multiple sensor data may be concatenated. Make sure the _data pointer is updated before getting Sensor Property ID of the corresponding sensor data.

Return Sensor Property ID of the sensor data.

Parameters

- _data: Pointer to the start of the sensor data.
- _fmt: Format of the sensor data.

ESP_BLE_MESH_SENSOR_DATA_FORMAT_A_MPID (*_len*, *_id*)

Generate a MPID value for sensor data with Format A.

Note 1. The Format field is 0b0 and indicates that Format A is used.

1. The Length field is a 1-based uint4 value (valid range 0x0–0xF, representing range of 1–16).
2. The Property ID is an 11-bit bit field representing 11 LSb of a Property ID.
3. This format may be used for Property Values that are not longer than 16 octets and for Property IDs less than 0x0800.

Return 2-octet MPID value for sensor data with Format A.

Parameters

- *_len*: Length of Sensor Raw value.
- *_id*: Sensor Property ID.

ESP_BLE_MESH_SENSOR_DATA_FORMAT_B_MPID (*_len*, *_id*)

Generate a MPID value for sensor data with Format B.

Note 1. The Format field is 0b1 and indicates Format B is used.

1. The Length field is a 1-based uint7 value (valid range 0x0–0x7F, representing range of 1–127). The value 0x7F represents a length of zero.
2. The Property ID is a 16-bit bit field representing a Property ID.
3. This format may be used for Property Values not longer than 128 octets and for any Property IDs. Property values longer than 128 octets are not supported by the Sensor Status message.
4. Exclude the generated 1-octet value, the 2-octet Sensor Property ID

Return 3-octet MPID value for sensor data with Format B.

Parameters

- *_len*: Length of Sensor Raw value.
- *_id*: Sensor Property ID.

Type Definitions

```
typedef void (*esp_ble_mesh_sensor_client_cb_t) (esp_ble_mesh_sensor_client_cb_event_t
                                                event, esp_ble_mesh_sensor_client_cb_param_t
                                                *param)
```

Bluetooth Mesh Sensor Client Model function.

Sensor Client Model callback function type

Parameters

- *event*: Event type
- *param*: Pointer to callback parameter

```
typedef void (*esp_ble_mesh_sensor_server_cb_t) (esp_ble_mesh_sensor_server_cb_event_t
                                                event, esp_ble_mesh_sensor_server_cb_param_t
                                                *param)
```

Bluetooth Mesh Sensor Server Model function.

Sensor Server Model callback function type

Parameters

- *event*: Event type
- *param*: Pointer to callback parameter

Enumerations

```
enum esp_ble_mesh_sensor_client_cb_event_t
```

This enum value is the event of Sensor Client Model

Values:

```
ESP_BLE_MESH_SENSOR_CLIENT_GET_STATE_EVT
```

```
ESP_BLE_MESH_SENSOR_CLIENT_SET_STATE_EVT
```

```
ESP_BLE_MESH_SENSOR_CLIENT_PUBLISH_EVT
```

```
ESP_BLE_MESH_SENSOR_CLIENT_TIMEOUT_EVT
```

ESP_BLE_MESH_SENSOR_CLIENT_EVT_MAX
enum esp_ble_mesh_sensor_sample_func
 This enum value is value of Sensor Sampling Function

Values:

ESP_BLE_MESH_SAMPLE_FUNC_UNSPECIFIED
ESP_BLE_MESH_SAMPLE_FUNC_INSTANTANEOUS
ESP_BLE_MESH_SAMPLE_FUNC_ARITHMETIC_MEAN
ESP_BLE_MESH_SAMPLE_FUNC_RMS
ESP_BLE_MESH_SAMPLE_FUNC_MAXIMUM
ESP_BLE_MESH_SAMPLE_FUNC_MINIMUM
ESP_BLE_MESH_SAMPLE_FUNC_ACCUMULATED
ESP_BLE_MESH_SAMPLE_FUNC_COUNT

enum esp_ble_mesh_sensor_server_cb_event_t
 This enum value is the event of Sensor Server Model

Values:

ESP_BLE_MESH_SENSOR_SERVER_STATE_CHANGE_EVT

1. When `get_auto_rsp` is set to `ESP_BLE_MESH_SERVER_AUTO_RSP`, no event will be callback to the application layer when Sensor Get messages are received.
2. When `set_auto_rsp` is set to `ESP_BLE_MESH_SERVER_AUTO_RSP`, this event will be callback to the application layer when Sensor Set/Set Unack messages are received.

ESP_BLE_MESH_SENSOR_SERVER_RECV_GET_MSG_EVT

When `get_auto_rsp` is set to `ESP_BLE_MESH_SERVER_RSP_BY_APP`, this event will be callback to the application layer when Sensor Get messages are received.

ESP_BLE_MESH_SENSOR_SERVER_RECV_SET_MSG_EVT

When `set_auto_rsp` is set to `ESP_BLE_MESH_SERVER_RSP_BY_APP`, this event will be callback to the application layer when Sensor Set/Set Unack messages are received.

ESP_BLE_MESH_SENSOR_SERVER_EVT_MAX

Lighting Client/Server Models

Header File

- `bt/esp_ble_mesh/api/models/include/esp_ble_mesh_time_scene_model_api.h`

Functions

`esp_err_t esp_ble_mesh_register_time_scene_client_callback` (`esp_ble_mesh_time_scene_client_cb_t callback`)

Register BLE Mesh Time Scene Client Model callback.

Return `ESP_OK` on success or error code otherwise.

Parameters

- `[in] callback`: Pointer to the callback function.

`esp_err_t esp_ble_mesh_time_scene_client_get_state` (`esp_ble_mesh_client_common_param_t *params`,
`esp_ble_mesh_time_scene_client_get_state_t *get_state`)

Get the value of Time Scene Server Model states using the Time Scene Client Model get messages.

Note If you want to know the opcodes and corresponding meanings accepted by this API, please refer to `esp_ble_mesh_time_scene_message_opcode_t` in `esp_ble_mesh_defs.h`

Return ESP_OK on success or error code otherwise.

Parameters

- [in] `params`: Pointer to BLE Mesh common client parameters.
- [in] `get_state`: Pointer to time scene get message value. Shall not be set to NULL.

```
esp_err_t esp_ble_mesh_time_scene_client_set_state(esp_ble_mesh_client_common_param_t
                                                *params,
                                                esp_ble_mesh_time_scene_client_set_state_t
                                                *set_state)
```

Set the value of Time Scene Server Model states using the Time Scene Client Model set messages.

Note If you want to know the opcodes and corresponding meanings accepted by this API, please refer to `esp_ble_mesh_time_scene_message_opcode_t` in `esp_ble_mesh_defs.h`

Return ESP_OK on success or error code otherwise.

Parameters

- [in] `params`: Pointer to BLE Mesh common client parameters.
- [in] `set_state`: Pointer to time scene set message value. Shall not be set to NULL.

```
esp_err_t esp_ble_mesh_register_time_scene_server_callback(esp_ble_mesh_time_scene_server_cb_t
                                                         callback)
```

Register BLE Mesh Time and Scenes Server Model callback.

Return ESP_OK on success or error code otherwise.

Parameters

- [in] `callback`: Pointer to the callback function.

Unions

```
union esp_ble_mesh_time_scene_client_get_state_t
#include <esp_ble_mesh_time_scene_model_api.h> Time Scene Client Model get message union.
```

Public Members

```
esp_ble_mesh_scheduler_act_get_t scheduler_act_get
For ESP_BLE_MESH_MODEL_OP_SCHEDULER_ACT_GET
union esp_ble_mesh_time_scene_client_set_state_t
#include <esp_ble_mesh_time_scene_model_api.h> Time Scene Client Model set message union.
```

Public Members

```
esp_ble_mesh_time_set_t time_set
For ESP_BLE_MESH_MODEL_OP_TIME_SET
esp_ble_mesh_time_zone_set_t time_zone_set
For ESP_BLE_MESH_MODEL_OP_TIME_ZONE_SET
esp_ble_mesh_tai_utc_delta_set_t tai_utc_delta_set
For ESP_BLE_MESH_MODEL_OP_TAI_UTC_DELTA_SET
esp_ble_mesh_time_role_set_t time_role_set
For ESP_BLE_MESH_MODEL_OP_TIME_ROLE_SET
esp_ble_mesh_scene_store_t scene_store
For ESP_BLE_MESH_MODEL_OP_SCENE_STORE & ESP_BLE_MESH_MODEL_OP_SCENE_STORE_UNACK
esp_ble_mesh_scene_recall_t scene_recall
For ESP_BLE_MESH_MODEL_OP_SCENE_RECALL & ESP_BLE_MESH_MODEL_OP_SCENE_RECALL_UNACK
esp_ble_mesh_scene_delete_t scene_delete
For ESP_BLE_MESH_MODEL_OP_SCENE_DELETE & ESP_BLE_MESH_MODEL_OP_SCENE_DELETE_UNACK
```

```
esp_ble_mesh_scheduler_act_set_t scheduler_act_set
    For ESP_BLE_MESH_MODEL_OP_SCHEDULER_ACT_SET &
    ESP_BLE_MESH_MODEL_OP_SCHEDULER_ACT_SET_UNACK
```

```
union esp_ble_mesh_time_scene_client_status_cb_t
    #include <esp_ble_mesh_time_scene_model_api.h> Time Scene Client Model received message union.
```

Public Members

```
esp_ble_mesh_time_status_cb_t time_status
    For ESP_BLE_MESH_MODEL_OP_TIME_STATUS

esp_ble_mesh_time_zone_status_cb_t time_zone_status
    For ESP_BLE_MESH_MODEL_OP_TIME_ZONE_STATUS

esp_ble_mesh_tai_utc_delta_status_cb_t tai_utc_delta_status
    For ESP_BLE_MESH_MODEL_OP_TAI_UTC_DELTA_STATUS

esp_ble_mesh_time_role_status_cb_t time_role_status
    For ESP_BLE_MESH_MODEL_OP_TIME_ROLE_STATUS

esp_ble_mesh_scene_status_cb_t scene_status
    For ESP_BLE_MESH_MODEL_OP_SCENE_STATUS

esp_ble_mesh_scene_register_status_cb_t scene_register_status
    For ESP_BLE_MESH_MODEL_OP_SCENE_REGISTER_STATUS

esp_ble_mesh_scheduler_status_cb_t scheduler_status
    For ESP_BLE_MESH_MODEL_OP_SCHEDULER_STATUS

esp_ble_mesh_scheduler_act_status_cb_t scheduler_act_status
    For ESP_BLE_MESH_MODEL_OP_SCHEDULER_ACT_STATUS
```

```
union esp_ble_mesh_time_scene_server_state_change_t
    #include <esp_ble_mesh_time_scene_model_api.h> Time Scene Server Model state change value union.
```

Public Members

```
esp_ble_mesh_state_change_time_set_t time_set
    The rcv_op in ctx can be used to decide which state is changed. Time Set

esp_ble_mesh_state_change_time_status_t time_status
    Time Status

esp_ble_mesh_state_change_time_zone_set_t time_zone_set
    Time Zone Set

esp_ble_mesh_state_change_tai_utc_delta_set_t tai_utc_delta_set
    TAI UTC Delta Set

esp_ble_mesh_state_change_time_role_set_t time_role_set
    Time Role Set

esp_ble_mesh_state_change_scene_store_t scene_store
    Scene Store

esp_ble_mesh_state_change_scene_recall_t scene_recall
    Scene Recall

esp_ble_mesh_state_change_scene_delete_t scene_delete
    Scene Delete

esp_ble_mesh_state_change_scheduler_act_set_t scheduler_act_set
    Scheduler Action Set
```

union esp_ble_mesh_time_scene_server_rcv_get_msg_t
#include <esp_ble_mesh_time_scene_model_api.h> Time Scene Server Model received get message union.

Public Members

esp_ble_mesh_server_rcv_scheduler_act_get_t **scheduler_act**
Scheduler Action Get

union esp_ble_mesh_time_scene_server_rcv_set_msg_t
#include <esp_ble_mesh_time_scene_model_api.h> Time Scene Server Model received set message union.

Public Members

esp_ble_mesh_server_rcv_time_set_t **time**
Time Set

esp_ble_mesh_server_rcv_time_zone_set_t **time_zone**
Time Zone Set

esp_ble_mesh_server_rcv_tai_utc_delta_set_t **tai_utc_delta**
TAI-UTC Delta Set

esp_ble_mesh_server_rcv_time_role_set_t **time_role**
Time Role Set

esp_ble_mesh_server_rcv_scene_store_t **scene_store**
Scene Store/Scene Store Unack

esp_ble_mesh_server_rcv_scene_recall_t **scene_recall**
Scene Recall/Scene Recall Unack

esp_ble_mesh_server_rcv_scene_delete_t **scene_delete**
Scene Delete/Scene Delete Unack

esp_ble_mesh_server_rcv_scheduler_act_set_t **scheduler_act**
Scheduler Action Set/Scheduler Action Set Unack

union esp_ble_mesh_time_scene_server_rcv_status_msg_t
#include <esp_ble_mesh_time_scene_model_api.h> Time Scene Server Model received status message union.

Public Members

esp_ble_mesh_server_rcv_time_status_t **time_status**
Time Status

union esp_ble_mesh_time_scene_server_cb_value_t
#include <esp_ble_mesh_time_scene_model_api.h> Time Scene Server Model callback value union.

Public Members

esp_ble_mesh_time_scene_server_state_change_t **state_change**
ESP_BLE_MESH_TIME_SCENE_SERVER_STATE_CHANGE_EVT

esp_ble_mesh_time_scene_server_rcv_get_msg_t **get**
ESP_BLE_MESH_TIME_SCENE_SERVER_RECV_GET_MSG_EVT

esp_ble_mesh_time_scene_server_rcv_set_msg_t **set**
ESP_BLE_MESH_TIME_SCENE_SERVER_RECV_SET_MSG_EVT

esp_ble_mesh_time_scene_server_rcv_status_msg_t **status**
ESP_BLE_MESH_TIME_SCENE_SERVER_RECV_STATUS_MSG_EVT

Structures

struct esp_ble_mesh_time_set_t

Bluetooth Mesh Time Scene Client Model Get and Set parameters structure.

Parameters of Time Set

Public Members

uint8_t tai_seconds[5]

The current TAI time in seconds

uint8_t sub_second

The sub-second time in units of 1/256 second

uint8_t uncertainty

The estimated uncertainty in 10-millisecond steps

uint16_t time_authority : 1

0 = No Time Authority, 1 = Time Authority

uint16_t tai_utc_delta : 15

Current difference between TAI and UTC in seconds

uint8_t time_zone_offset

The local time zone offset in 15-minute increments

struct esp_ble_mesh_time_zone_set_t

Parameters of Time Zone Set

Public Members

uint8_t time_zone_offset_new

Upcoming local time zone offset

uint8_t tai_zone_change[5]

TAI Seconds time of the upcoming Time Zone Offset change

struct esp_ble_mesh_tai_utc_delta_set_t

Parameters of TAI-UTC Delta Set

Public Members

uint16_t tai_utc_delta_new : 15

Upcoming difference between TAI and UTC in seconds

uint16_t padding : 1

Always 0b0. Other values are Prohibited.

uint8_t tai_delta_change[5]

TAI Seconds time of the upcoming TAI-UTC Delta change

struct esp_ble_mesh_time_role_set_t

Parameter of Time Role Set

Public Members

uint8_t time_role

The Time Role for the element

struct esp_ble_mesh_scene_store_t

Parameter of Scene Store

Public Members

`uint16_t scene_number`
The number of scenes to be stored

struct esp_ble_mesh_scene_recall_t
Parameters of Scene Recall

Public Members

`bool op_en`
Indicate if optional parameters are included

`uint16_t scene_number`
The number of scenes to be recalled

`uint8_t tid`
Transaction ID

`uint8_t trans_time`
Time to complete state transition (optional)

`uint8_t delay`
Indicate message execution delay (C.1)

struct esp_ble_mesh_scene_delete_t
Parameter of Scene Delete

Public Members

`uint16_t scene_number`
The number of scenes to be deleted

struct esp_ble_mesh_scheduler_act_get_t
Parameter of Scheduler Action Get

Public Members

`uint8_t index`
Index of the Schedule Register entry to get

struct esp_ble_mesh_scheduler_act_set_t
Parameters of Scheduler Action Set

Public Members

`uint64_t index : 4`
Index of the Schedule Register entry to set

`uint64_t year : 7`
Scheduled year for the action

`uint64_t month : 12`
Scheduled month for the action

`uint64_t day : 5`
Scheduled day of the month for the action

`uint64_t hour : 5`
Scheduled hour for the action

`uint64_t minute` : 6
Scheduled minute for the action

`uint64_t second` : 6
Scheduled second for the action

`uint64_t day_of_week` : 7
Schedule days of the week for the action

`uint64_t action` : 4
Action to be performed at the scheduled time

`uint64_t trans_time` : 8
Transition time for this action

`uint16_t scene_number`
Transition time for this action

struct esp_ble_mesh_time_status_cb_t
Bluetooth Mesh Time Scene Client Model Get and Set callback parameters structure.
Parameters of Time Status

Public Members

`uint8_t tai_seconds`[5]
The current TAI time in seconds

`uint8_t sub_second`
The sub-second time in units of 1/256 second

`uint8_t uncertainty`
The estimated uncertainty in 10-millisecond steps

`uint16_t time_authority` : 1
0 = No Time Authority, 1 = Time Authority

`uint16_t tai_utc_delta` : 15
Current difference between TAI and UTC in seconds

`uint8_t time_zone_offset`
The local time zone offset in 15-minute increments

struct esp_ble_mesh_time_zone_status_cb_t
Parameters of Time Zone Status

Public Members

`uint8_t time_zone_offset_curr`
Current local time zone offset

`uint8_t time_zone_offset_new`
Upcoming local time zone offset

`uint8_t tai_zone_change`[5]
TAI Seconds time of the upcoming Time Zone Offset change

struct esp_ble_mesh_tai_utc_delta_status_cb_t
Parameters of TAI-UTC Delta Status

Public Members

`uint16_t tai_utc_delta_curr` : 15
Current difference between TAI and UTC in seconds

`uint16_t padding_1 : 1`
Always 0b0. Other values are Prohibited.

`uint16_t tai_utc_delta_new : 15`
Upcoming difference between TAI and UTC in seconds

`uint16_t padding_2 : 1`
Always 0b0. Other values are Prohibited.

`uint8_t tai_delta_change[5]`
TAI Seconds time of the upcoming TAI-UTC Delta change

struct esp_ble_mesh_time_role_status_cb_t
Parameter of Time Role Status

Public Members

`uint8_t time_role`
The Time Role for the element

struct esp_ble_mesh_scene_status_cb_t
Parameters of Scene Status

Public Members

`bool op_en`
Indicate if optional parameters are included

`uint8_t status_code`
Status code of the last operation

`uint16_t current_scene`
Scene Number of the current scene

`uint16_t target_scene`
Scene Number of the target scene (optional)

`uint8_t remain_time`
Time to complete state transition (C.1)

struct esp_ble_mesh_scene_register_status_cb_t
Parameters of Scene Register Status

Public Members

`uint8_t status_code`
Status code for the previous operation

`uint16_t current_scene`
Scene Number of the current scene

struct net_buf_simple *scenes
A list of scenes stored within an element

struct esp_ble_mesh_scheduler_status_cb_t
Parameter of Scheduler Status

Public Members

`uint16_t schedules`
Bit field indicating defined Actions in the Schedule Register

struct esp_ble_mesh_scheduler_act_status_cb_t
Parameters of Scheduler Action Status

Public Members

uint64_t **index** : 4
Enumerates (selects) a Schedule Register entry

uint64_t **year** : 7
Scheduled year for the action

uint64_t **month** : 12
Scheduled month for the action

uint64_t **day** : 5
Scheduled day of the month for the action

uint64_t **hour** : 5
Scheduled hour for the action

uint64_t **minute** : 6
Scheduled minute for the action

uint64_t **second** : 6
Scheduled second for the action

uint64_t **day_of_week** : 7
Schedule days of the week for the action

uint64_t **action** : 4
Action to be performed at the scheduled time

uint64_t **trans_time** : 8
Transition time for this action

uint16_t **scene_number**
Transition time for this action

struct esp_ble_mesh_time_scene_client_cb_param_t
Time Scene Client Model callback parameters

Public Members

int **error_code**
Appropriate error code

esp_ble_mesh_client_common_param_t ***params**
The client common parameters.

esp_ble_mesh_time_scene_client_status_cb_t **status_cb**
The scene status message callback values

struct esp_ble_mesh_time_state_t
Parameters of Time state

Public Members

uint8_t **tai_seconds**[5]
The value of the TAI Seconds state

uint8_t **subsecond**
The value of the Subsecond field

uint8_t uncertainty
The value of the Uncertainty field

uint8_t time_zone_offset_curr
The value of the Time Zone Offset Current field

uint8_t time_zone_offset_new
The value of the Time Zone Offset New state

uint8_t tai_zone_change[5]
The value of the TAI of Zone Chaneg field

uint16_t time_authority : 1
The value of the Time Authority bit

uint16_t tai_utc_delta_curr : 15
The value of the TAI-UTC Delta Current state

uint16_t tai_utc_delta_new : 15
The value of the TAI-UTC Delta New state

uint8_t tai_delta_change[5]
The value of the TAI of Delta Change field

struct esp_ble_mesh_time_state_t::[anonymous] time
Parameters of the Time state

uint8_t time_role
The value of the Time Role state

struct esp_ble_mesh_time_srv_t
User data of Time Server Model

Public Members

esp_ble_mesh_model_t *model
Pointer to the Time Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t rsp_ctrl
Response control of the server model received messages

esp_ble_mesh_time_state_t *state
Parameters of the Time state

struct esp_ble_mesh_time_setup_srv_t
User data of Time Setup Server Model

Public Members

esp_ble_mesh_model_t *model
Pointer to the Time Setup Server Model. Initialized internally.

esp_ble_mesh_server_rsp_ctrl_t rsp_ctrl
Response control of the server model received messages

esp_ble_mesh_time_state_t *state
Parameters of the Time state

struct esp_ble_mesh_scene_register_t

1. Scene Store is an operation of storing values of a present state of an element.
2. The structure and meaning of the stored state is determined by a model. States to be stored are specified by each model.
3. The Scene Store operation shall persistently store all values of all states marked as Stored with Scene for all models present on all elements of a node.

4. If a model is extending another model, the extending model shall determine the Stored with Scene behavior of that model. Parameters of Scene Register state

Public Members

`uint16_t scene_number`

The value of the Scene Number

`uint8_t scene_type`

The value of the Scene Type

`struct net_buf_simple *scene_value`

Scene value may use a union to represent later, the union contains structures of all the model states which can be stored in a scene. The value of the Scene Value

`struct esp_ble_mesh_scenes_state_t`

Parameters of Scenes state.

Scenes serve as memory banks for storage of states (e.g., a power level or a light level/color). Values of states of an element can be stored as a scene and can be recalled later from the scene memory.

A scene is represented by a Scene Number, which is a 16-bit non-zero, mesh-wide value. (There can be a maximum of 65535 scenes in a mesh network.) The meaning of a scene, as well as the state storage container associated with it, are determined by a model.

The Scenes state change may start numerous parallel model transitions. In that case, each individual model handles the transition internally.

The scene transition is defined as a group of individual model transitions started by a Scene Recall operation. The scene transition is in progress when at least one transition from the group of individual model transitions is in progress.

Public Members

`const uint16_t scene_count`

The Scenes state's scene count

`esp_ble_mesh_scene_register_t *scenes`

Parameters of the Scenes state

`uint16_t current_scene`

The Current Scene state is a 16-bit value that contains either the Scene Number of the currently active scene or a value of 0x0000 when no scene is active.

When a Scene Store operation or a Scene Recall operation completes with success, the Current Scene state value shall be to the Scene Number used during that operation.

When the Current Scene Number is deleted from a Scene Register state as a result of Scene Delete operation, the Current Scene state shall be set to 0x0000.

When any of the element's state that is marked as "Stored with Scene" has changed not as a result of a Scene Recall operation, the value of the Current Scene state shall be set to 0x0000.

When a scene transition is in progress, the value of the Current Scene state shall be set to 0x0000. The value of the Current Scene state

`uint16_t target_scene`

The Target Scene state is a 16-bit value that contains the target Scene Number when a scene transition is in progress.

When the scene transition is in progress and the target Scene Number is deleted from a Scene Register state as a result of Scene Delete operation, the Target Scene state shall be set to 0x0000.

When the scene transition is in progress and a new Scene Number is stored in the Scene Register as a result of Scene Store operation, the Target Scene state shall be set to the new Scene Number.

When the scene transition is not in progress, the value of the Target Scene state shall be set to 0x0000. The value of the Target Scene state

`uint8_t status_code`

The status code of the last scene operation

`bool in_progress`

Indicate if the scene transition is in progress

`struct esp_ble_mesh_scene_srv_t`

User data of Scene Server Model

Public Members

`esp_ble_mesh_model_t *model`

Pointer to the Scene Server Model. Initialized internally.

`esp_ble_mesh_server_rsp_ctrl_t rsp_ctrl`

Response control of the server model received messages

`esp_ble_mesh_scenes_state_t *state`

Parameters of the Scenes state

`esp_ble_mesh_last_msg_info_t last`

Parameters of the last received set message

`esp_ble_mesh_state_transition_t transition`

Parameters of state transition

`struct esp_ble_mesh_scene_setup_srv_t`

User data of Scene Setup Server Model

Public Members

`esp_ble_mesh_model_t *model`

Pointer to the Scene Setup Server Model. Initialized internally.

`esp_ble_mesh_server_rsp_ctrl_t rsp_ctrl`

Response control of the server model received messages

`esp_ble_mesh_scenes_state_t *state`

Parameters of the Scenes state

`struct esp_ble_mesh_schedule_register_t`

Parameters of Scheduler Register state

Public Members

`bool in_use`

Indicate if the registered schedule is in use

`uint64_t year : 7`

The value of Scheduled year for the action

`uint64_t month : 12`

The value of Scheduled month for the action

`uint64_t day : 5`

The value of Scheduled day of the month for the action

`uint64_t hour : 5`

The value of Scheduled hour for the action

`uint64_t minute` : 6
The value of Scheduled minute for the action

`uint64_t second` : 6
The value of Scheduled second for the action

`uint64_t day_of_week` : 7
The value of Schedule days of the week for the action

`uint64_t action` : 4
The value of Action to be performed at the scheduled time

`uint64_t trans_time` : 8
The value of Transition time for this action

`uint16_t scene_number`
The value of Scene Number to be used for some actions

struct esp_ble_mesh_scheduler_state_t
Parameters of Scheduler state

Public Members

`const uint8_t schedule_count`
Scheduler count

`esp_ble_mesh_schedule_register_t *schedules`
Up to 16 scheduled entries

struct esp_ble_mesh_scheduler_srv_t
User data of Scheduler Server Model

Public Members

`esp_ble_mesh_model_t *model`
Pointer to the Scheduler Server Model. Initialized internally.

`esp_ble_mesh_server_rsp_ctrl_t rsp_ctrl`
Response control of the server model received messages

`esp_ble_mesh_scheduler_state_t *state`
Parameters of the Scheduler state

struct esp_ble_mesh_scheduler_setup_srv_t
User data of Scheduler Setup Server Model

Public Members

`esp_ble_mesh_model_t *model`
Pointer to the Scheduler Setup Server Model. Initialized internally.

`esp_ble_mesh_server_rsp_ctrl_t rsp_ctrl`
Response control of the server model received messages

`esp_ble_mesh_scheduler_state_t *state`
Parameters of the Scheduler state

struct esp_ble_mesh_state_change_time_set_t
Parameters of Time Set state change event

Public Members

`uint8_t tai_seconds[5]`
The current TAI time in seconds

`uint8_t subsecond`
The sub-second time in units of 1/256 second

`uint8_t uncertainty`
The estimated uncertainty in 10-millisecond steps

`uint16_t time_authority : 1`
0 = No Time Authority, 1 = Time Authority

`uint16_t tai_utc_delta_curr : 15`
Current difference between TAI and UTC in seconds

`uint8_t time_zone_offset_curr`
The local time zone offset in 15-minute increments

struct esp_ble_mesh_state_change_time_status_t
Parameters of Time Status state change event

Public Members

`uint8_t tai_seconds[5]`
The current TAI time in seconds

`uint8_t subsecond`
The sub-second time in units of 1/256 second

`uint8_t uncertainty`
The estimated uncertainty in 10-millisecond steps

`uint16_t time_authority : 1`
0 = No Time Authority, 1 = Time Authority

`uint16_t tai_utc_delta_curr : 15`
Current difference between TAI and UTC in seconds

`uint8_t time_zone_offset_curr`
The local time zone offset in 15-minute increments

struct esp_ble_mesh_state_change_time_zone_set_t
Parameters of Time Zone Set state change event

Public Members

`uint8_t time_zone_offset_new`
Upcoming local time zone offset

`uint8_t tai_zone_change[5]`
TAI Seconds time of the upcoming Time Zone Offset change

struct esp_ble_mesh_state_change_tai_utc_delta_set_t
Parameters of TAI UTC Delta Set state change event

Public Members

`uint16_t tai_utc_delta_new : 15`
Upcoming difference between TAI and UTC in seconds

`uint8_t tai_delta_change[5]`
TAI Seconds time of the upcoming TAI-UTC Delta change

`struct esp_ble_mesh_state_change_time_role_set_t`
Parameter of Time Role Set state change event

Public Members

`uint8_t time_role`
The Time Role for the element

`struct esp_ble_mesh_state_change_scene_store_t`
Parameter of Scene Store state change event

Public Members

`uint16_t scene_number`
The number of scenes to be stored

`struct esp_ble_mesh_state_change_scene_recall_t`
Parameter of Scene Recall state change event

Public Members

`uint16_t scene_number`
The number of scenes to be recalled

`struct esp_ble_mesh_state_change_scene_delete_t`
Parameter of Scene Delete state change event

Public Members

`uint16_t scene_number`
The number of scenes to be deleted

`struct esp_ble_mesh_state_change_scheduler_act_set_t`
Parameter of Scheduler Action Set state change event

Public Members

`uint64_t index : 4`
Index of the Schedule Register entry to set

`uint64_t year : 7`
Scheduled year for the action

`uint64_t month : 12`
Scheduled month for the action

`uint64_t day : 5`
Scheduled day of the month for the action

`uint64_t hour : 5`
Scheduled hour for the action

`uint64_t minute : 6`
Scheduled minute for the action

`uint64_t second : 6`
Scheduled second for the action

`uint64_t day_of_week` : 7
Schedule days of the week for the action

`uint64_t action` : 4
Action to be performed at the scheduled time

`uint64_t trans_time` : 8
Transition time for this action

`uint16_t scene_number`
Scene number to be used for some actions

struct esp_ble_mesh_server_rcv_scheduler_act_get_t
Context of the received Scheduler Action Get message

Public Members

`uint8_t index`
Index of the Schedule Register entry to get

struct esp_ble_mesh_server_rcv_time_set_t
Context of the received Time Set message

Public Members

`uint8_t tai_seconds`[5]
The current TAI time in seconds

`uint8_t subsecond`
The sub-second time in units of 1/256 second

`uint8_t uncertainty`
The estimated uncertainty in 10-millisecond steps

`uint16_t time_authority` : 1
0 = No Time Authority, 1 = Time Authority

`uint16_t tai_utc_delta` : 15
Current difference between TAI and UTC in seconds

`uint8_t time_zone_offset`
The local time zone offset in 15-minute increments

struct esp_ble_mesh_server_rcv_time_zone_set_t
Context of the received Time Zone Set message

Public Members

`uint8_t time_zone_offset_new`
Upcoming local time zone offset

`uint8_t tai_zone_change`[5]
TAI Seconds time of the upcoming Time Zone Offset change

struct esp_ble_mesh_server_rcv_tai_utc_delta_set_t
Context of the received TAI UTC Delta Set message

Public Members

`uint16_t tai_utc_delta_new` : 15
Upcoming difference between TAI and UTC in seconds

`uint16_t padding : 1`
Always 0b0. Other values are Prohibited.

`uint8_t tai_delta_change[5]`
TAI Seconds time of the upcoming TAI-UTC Delta change

struct esp_ble_mesh_server_rcv_time_role_set_t
Context of the received Time Role Set message

Public Members

`uint8_t time_role`
The Time Role for the element

struct esp_ble_mesh_server_rcv_scene_store_t
Context of the received Scene Store message

Public Members

`uint16_t scene_number`
The number of scenes to be stored

struct esp_ble_mesh_server_rcv_scene_recall_t
Context of the received Scene Recall message

Public Members

`bool op_en`
Indicate if optional parameters are included

`uint16_t scene_number`
The number of scenes to be recalled

`uint8_t tid`
Transaction ID

`uint8_t trans_time`
Time to complete state transition (optional)

`uint8_t delay`
Indicate message execution delay (C.1)

struct esp_ble_mesh_server_rcv_scene_delete_t
Context of the received Scene Delete message

Public Members

`uint16_t scene_number`
The number of scenes to be deleted

struct esp_ble_mesh_server_rcv_scheduler_act_set_t
Context of the received Scheduler Action Set message

Public Members

`uint64_t index : 4`
Index of the Schedule Register entry to set

`uint64_t year : 7`
Scheduled year for the action

`uint64_t month` : 12
Scheduled month for the action

`uint64_t day` : 5
Scheduled day of the month for the action

`uint64_t hour` : 5
Scheduled hour for the action

`uint64_t minute` : 6
Scheduled minute for the action

`uint64_t second` : 6
Scheduled second for the action

`uint64_t day_of_week` : 7
Schedule days of the week for the action

`uint64_t action` : 4
Action to be performed at the scheduled time

`uint64_t trans_time` : 8
Transition time for this action

`uint16_t scene_number`
Scene number to be used for some actions

struct esp_ble_mesh_server_rcv_time_status_t
Context of the received Time Status message

Public Members

`uint8_t tai_seconds`[5]
The current TAI time in seconds

`uint8_t subsecond`
The sub-second time in units of 1/256 second

`uint8_t uncertainty`
The estimated uncertainty in 10-millisecond steps

`uint16_t time_authority` : 1
0 = No Time Authority, 1 = Time Authority

`uint16_t tai_utc_delta` : 15
Current difference between TAI and UTC in seconds

`uint8_t time_zone_offset`
The local time zone offset in 15-minute increments

struct esp_ble_mesh_time_scene_server_cb_param_t
Time Scene Server Model callback parameters

Public Members

`esp_ble_mesh_model_t *model`
Pointer to Time and Scenes Server Models

`esp_ble_mesh_msg_ctx_t ctx`
Context of the received messages

`esp_ble_mesh_time_scene_server_cb_value_t value`
Value of the received Time and Scenes Messages

Macros**ESP_BLE_MESH_MODEL_TIME_CLI** (cli_pub, cli_data)

Define a new Time Client Model.

Note This API needs to be called for each element on which the application needs to have a Time Client Model.**Return** New Time Client Model instance.**Parameters**

- cli_pub: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- cli_data: Pointer to the unique struct *esp_ble_mesh_client_t*.

ESP_BLE_MESH_MODEL_SCENE_CLI (cli_pub, cli_data)

Define a new Scene Client Model.

Note This API needs to be called for each element on which the application needs to have a Scene Client Model.**Return** New Scene Client Model instance.**Parameters**

- cli_pub: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- cli_data: Pointer to the unique struct *esp_ble_mesh_client_t*.

ESP_BLE_MESH_MODEL_SCHEDULER_CLI (cli_pub, cli_data)

Define a new Scheduler Client Model.

Note This API needs to be called for each element on which the application needs to have a Scheduler Client Model.**Return** New Scheduler Client Model instance.**Parameters**

- cli_pub: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- cli_data: Pointer to the unique struct *esp_ble_mesh_client_t*.

ESP_BLE_MESH_MODEL_TIME_SRV (srv_pub, srv_data)

Time Scene Server Models related context.

Define a new Time Server Model.

Note 1. The Time Server model is a root model. When this model is present on an Element, the corresponding Time Setup Server model shall also be present.

1. This model shall support model publication and model subscription.

Return New Time Server Model instance.**Parameters**

- srv_pub: Pointer to the unique struct *esp_ble_mesh_model_pub_t*.
- srv_data: Pointer to the unique struct *esp_ble_mesh_time_srv_t*.

ESP_BLE_MESH_MODEL_TIME_SETUP_SRV (srv_data)

Define a new Time Setup Server Model.

Note 1. The Time Setup Server model extends the Time Server model. Time is sensitive information that is propagated across a mesh network.

1. Only an authorized Time Client should be allowed to change the Time and Time Role states. A dedicated application key Bluetooth SIG Proprietary should be used on the Time Setup Server to restrict access to the server to only authorized Time Clients.
2. This model does not support subscribing nor publishing.

Return New Time Setup Server Model instance.**Parameters**

- srv_data: Pointer to the unique struct *esp_ble_mesh_time_setup_srv_t*.

ESP_BLE_MESH_MODEL_SCENE_SRV (srv_pub, srv_data)

Define a new Scene Server Model.

Note 1. The Scene Server model is a root model. When this model is present on an Element, the corresponding Scene Setup Server model shall also be present.

1. This model shall support model publication and model subscription.
2. The model may be present only on the Primary element of a node.

Return New Scene Server Model instance.

Parameters

- `srv_pub`: Pointer to the unique struct `esp_ble_mesh_model_pub_t`.
- `srv_data`: Pointer to the unique struct `esp_ble_mesh_scene_srv_t`.

ESP_BLE_MESH_MODEL_SCENE_SETUP_SRV (`srv_pub`, `srv_data`)

Define a new Scene Setup Server Model.

Note 1. The Scene Setup Server model extends the Scene Server model and the Generic Default Transition Time Server model.

1. This model shall support model subscription.
2. The model may be present only on the Primary element of a node.

Return New Scene Setup Server Model instance.

Parameters

- `srv_pub`: Pointer to the unique struct `esp_ble_mesh_model_pub_t`.
- `srv_data`: Pointer to the unique struct `esp_ble_mesh_scene_setup_srv_t`.

ESP_BLE_MESH_MODEL_SCHEDULER_SRV (`srv_pub`, `srv_data`)

Define a new Scheduler Server Model.

Note 1. The Scheduler Server model extends the Scene Server model. When this model is present on an Element, the corresponding Scheduler Setup Server model shall also be present.

1. This model shall support model publication and model subscription.
2. The model may be present only on the Primary element of a node.
3. The model requires the Time Server model shall be present on the element.

Return New Scheduler Server Model instance.

Parameters

- `srv_pub`: Pointer to the unique struct `esp_ble_mesh_model_pub_t`.
- `srv_data`: Pointer to the unique struct `esp_ble_mesh_scheduler_srv_t`.

ESP_BLE_MESH_MODEL_SCHEDULER_SETUP_SRV (`srv_pub`, `srv_data`)

Define a new Scheduler Setup Server Model.

Note 1. The Scheduler Setup Server model extends the Scheduler Server and the Scene Setup Server models.

1. This model shall support model subscription.
2. The model may be present only on the Primary element of a node.

Return New Scheduler Setup Server Model instance.

Parameters

- `srv_pub`: Pointer to the unique struct `esp_ble_mesh_model_pub_t`.
- `srv_data`: Pointer to the unique struct `esp_ble_mesh_scheduler_setup_srv_t`.

ESP_BLE_MESH_UNKNOWN_TAI_SECONDS

Unknown TAI Seconds

ESP_BLE_MESH_UNKNOWN_TAI_ZONE_CHANGE

Unknown TAI of Zone Change

ESP_BLE_MESH_UNKNOWN_TAI_DELTA_CHANGE

Unknown TAI of Delta Change

ESP_BLE_MESH_TAI_UTC_DELTA_MAX_VALUE

Maximum TAI-UTC Delta value

ESP_BLE_MESH_TAI_SECONDS_LEN

Length of TAI Seconds

ESP_BLE_MESH_TAI_OF_ZONE_CHANGE_LEN

Length of TAI of Zone Change

ESP_BLE_MESH_TAI_OF_DELTA_CHANGE_LEN

Length of TAI of Delta Change

ESP_BLE_MESH_INVALID_SCENE_NUMBER

Invalid Scene Number

ESP_BLE_MESH_SCENE_NUMBER_LEN

Length of the Scene Number

ESP_BLE_MESH_SCHEDULE_YEAR_ANY_YEAR
Any year of the Scheduled year

ESP_BLE_MESH_SCHEDULE_DAY_ANY_DAY
Any day of the Scheduled day

ESP_BLE_MESH_SCHEDULE_HOUR_ANY_HOUR
Any hour of the Scheduled hour

ESP_BLE_MESH_SCHEDULE_HOUR_ONCE_A_DAY
Any hour of the Scheduled Day

ESP_BLE_MESH_SCHEDULE_SEC_ANY_OF_HOUR
Any minute of the Scheduled hour

ESP_BLE_MESH_SCHEDULE_SEC EVERY_15_MIN
Every 15 minutes of the Scheduled hour

ESP_BLE_MESH_SCHEDULE_SEC EVERY_20_MIN
Every 20 minutes of the Scheduled hour

ESP_BLE_MESH_SCHEDULE_SEC_ONCE_AN_HOUR
Once of the Scheduled hour

ESP_BLE_MESH_SCHEDULE_SEC_ANY_OF_MIN
Any second of the Scheduled minute

ESP_BLE_MESH_SCHEDULE_SEC EVERY_15_SEC
Every 15 seconds of the Scheduled minute

ESP_BLE_MESH_SCHEDULE_SEC EVERY_20_SEC
Every 20 seconds of the Scheduled minute

ESP_BLE_MESH_SCHEDULE_SEC_ONCE_AN_MIN
Once of the Scheduled minute

ESP_BLE_MESH_SCHEDULE_ACT_TURN_OFF
Scheduled Action - Turn Off

ESP_BLE_MESH_SCHEDULE_ACT_TURN_ON
Scheduled Action - Turn On

ESP_BLE_MESH_SCHEDULE_ACT_SCENE_RECALL
Scheduled Action - Scene Recall

ESP_BLE_MESH_SCHEDULE_ACT_NO_ACTION
Scheduled Action - No Action

ESP_BLE_MESH_SCHEDULE_SCENE_NO_SCENE
Scheduled Scene - No Scene

ESP_BLE_MESH_SCHEDULE_ENTRY_MAX_INDEX
Maximum number of Scheduled entries

ESP_BLE_MESH_TIME_NONE
Time Role - None

ESP_BLE_MESH_TIME_AUTHORITY
Time Role - Mesh Time Authority

ESP_BLE_MESH_TIME_RELAY
Time Role - Mesh Time Relay

ESP_BLE_MESH_TIME_CLINET
Time Role - Mesh Time Client

ESP_BLE_MESH_SCENE_SUCCESS
Scene operation - Success

ESP_BLE_MESH_SCENE_REG_FULL
Scene operation - Scene Register Full

ESP_BLE_MESH_SCENE_NOT_FOUND
Scene operation - Scene Not Found

Type Definitions

```
typedef void (*esp_ble_mesh_time_scene_client_cb_t) (esp_ble_mesh_time_scene_client_cb_event_t  
event,  
esp_ble_mesh_time_scene_client_cb_param_t  
*param)
```

Bluetooth Mesh Time Scene Client Model function.

Time Scene Client Model callback function type

Parameters

- event: Event type
- param: Pointer to callback parameter

```
typedef void (*esp_ble_mesh_time_scene_server_cb_t) (esp_ble_mesh_time_scene_server_cb_event_t  
event,  
esp_ble_mesh_time_scene_server_cb_param_t  
*param)
```

Bluetooth Mesh Time and Scenes Server Model function.

Time Scene Server Model callback function type

Parameters

- event: Event type
- param: Pointer to callback parameter

Enumerations

```
enum esp_ble_mesh_time_scene_client_cb_event_t  
This enum value is the event of Time Scene Client Model
```

Values:

ESP_BLE_MESH_TIME_SCENE_CLIENT_GET_STATE_EVT

ESP_BLE_MESH_TIME_SCENE_CLIENT_SET_STATE_EVT

ESP_BLE_MESH_TIME_SCENE_CLIENT_PUBLISH_EVT

ESP_BLE_MESH_TIME_SCENE_CLIENT_TIMEOUT_EVT

ESP_BLE_MESH_TIME_SCENE_CLIENT_EVT_MAX

```
enum esp_ble_mesh_time_scene_server_cb_event_t  
This enum value is the event of Time Scene Server Model
```

Values:

ESP_BLE_MESH_TIME_SCENE_SERVER_STATE_CHANGE_EVT

1. When `get_auto_rsp` is set to `ESP_BLE_MESH_SERVER_AUTO_RSP`, no event will be callback to the application layer when Time Scene Get messages are received.
2. When `set_auto_rsp` is set to `ESP_BLE_MESH_SERVER_AUTO_RSP`, this event will be callback to the application layer when Time Scene Set/Set Unack messages are received.

ESP_BLE_MESH_TIME_SCENE_SERVER_RECV_GET_MSG_EVT

When `get_auto_rsp` is set to `ESP_BLE_MESH_SERVER_RSP_BY_APP`, this event will be callback to the application layer when Time Scene Get messages are received.

ESP_BLE_MESH_TIME_SCENE_SERVER_RECV_SET_MSG_EVT

When `set_auto_rsp` is set to `ESP_BLE_MESH_SERVER_RSP_BY_APP`, this event will be callback to the application layer when Time Scene Set/Set Unack messages are received.

ESP_BLE_MESH_TIME_SCENE_SERVER_RECV_STATUS_MSG_EVT

When `status_auto_rsp` is set to `ESP_BLE_MESH_SERVER_RSP_BY_APP`, this event will be callback to the application layer when Time Status message is received.

ESP_BLE_MESH_TIME_SCENE_SERVER_EVT_MAX

ESP-IDF currently supports two host stacks. The Blueroid based stack (default) supports classic Bluetooth as well as BLE. On the other hand, Apache NimBLE based stack is BLE only. For users to make a choice: * For usecases involving classic Bluetooth as well as BLE, Blueroid should be used. * For BLE-only usecases, using NimBLE is recommended. It is less demanding in terms of code footprint and runtime memory, making it suitable for such scenarios.

For the overview of the ESP32 Bluetooth stack architecture, follow the links below:

- [ESP32 Bluetooth Architecture \(PDF\) \[English\]](#)
- [ESP32 Bluetooth Architecture \(PDF\) \[中文\]](#)

Code examples for this API section are provided in the [bluetooth/bluedroid](#) directory of ESP-IDF examples.

The following examples contain detailed walkthroughs:

- [GATT Client Example Walkthrough](#)
- [GATT Server Service Table Example Walkthrough](#)
- [GATT Server Example Walkthrough](#)
- [GATT Security Client Example Walkthrough](#)
- [GATT Security Server Example Walkthrough](#)
- [GATT Client Multi-connection Example Walkthrough](#)

2.2 Networking APIs

2.2.1 Wi-Fi

Wi-Fi

Introduction The WiFi libraries provide support for configuring and monitoring the ESP32 WiFi networking functionality. This includes configuration for:

- Station mode (aka STA mode or WiFi client mode). ESP32 connects to an access point.
- AP mode (aka Soft-AP mode or Access Point mode). Stations connect to the ESP32.
- Combined AP-STA mode (ESP32 is concurrently an access point and a station connected to another access point).
- Various security modes for the above (WPA, WPA2, WEP, etc.)
- Scanning for access points (active & passive scanning).
- Promiscuous mode for monitoring of IEEE802.11 WiFi packets.

Application Examples The [wifi](#) directory of ESP-IDF examples contains the following applications:

Code examples for Wi-Fi are provided in the [wifi](#) directory of ESP-IDF examples.

In addition, there is a simple [esp-idf-template](#) application to demonstrate a minimal IDF project structure.

API Reference

Header File

- [esp_wifi/include/esp_wifi.h](#)

Functions

esp_err_t **esp_wifi_init** (*const wifi_init_config_t *config*)

Initialize WiFi Allocate resource for WiFi driver, such as WiFi control structure, RX/TX buffer, WiFi NVS structure etc. This WiFi also starts WiFi task.

Attention 1. This API must be called before all other WiFi API can be called

Attention 2. Always use WIFI_INIT_CONFIG_DEFAULT macro to initialize the configuration to default values, this can guarantee all the fields get correct value when more fields are added into *wifi_init_config_t* in future release. If you want to set your own initial values, overwrite the default values which are set by WIFI_INIT_CONFIG_DEFAULT. Please be notified that the field ‘magic’ of *wifi_init_config_t* should always be WIFI_INIT_CONFIG_MAGIC!

Return

- ESP_OK: succeed
- ESP_ERR_NO_MEM: out of memory
- others: refer to error code esp_err.h

Parameters

- config: pointer to WiFi initialized configuration structure; can point to a temporary variable.

esp_err_t **esp_wifi_deinit** (void)

Deinit WiFi Free all resource allocated in esp_wifi_init and stop WiFi task.

Attention 1. This API should be called if you want to remove WiFi driver from the system

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init

esp_err_t **esp_wifi_set_mode** (*wifi_mode_t mode*)

Set the WiFi operating mode.

Set the WiFi operating mode as station, soft-AP or station+soft-AP, The default mode is soft-AP mode.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument
- others: refer to error code in esp_err.h

Parameters

- mode: WiFi operating mode

esp_err_t **esp_wifi_get_mode** (*wifi_mode_t *mode*)

Get current operating mode of WiFi.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument

Parameters

- [out] mode: store current WiFi mode

esp_err_t **esp_wifi_start** (void)

Start WiFi according to current configuration If mode is WIFI_MODE_STA, it create station control block and start station If mode is WIFI_MODE_AP, it create soft-AP control block and start soft-AP If mode is WIFI_MODE_APSTA, it create soft-AP and station control block and start soft-AP and station.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument
- ESP_ERR_NO_MEM: out of memory
- ESP_ERR_WIFI_CONN: WiFi internal error, station or soft-AP control block wrong
- ESP_FAIL: other WiFi internal errors

esp_err_t **esp_wifi_stop** (void)

Stop WiFi If mode is WIFI_MODE_STA, it stop station and free station control block If mode is

WIFI_MODE_AP, it stop soft-AP and free soft-AP control block If mode is WIFI_MODE_APSTA, it stop station/soft-AP and free station/soft-AP control block.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init

esp_err_t **esp_wifi_restore** (void)

Restore WiFi stack persistent settings to default values.

This function will reset settings made using the following APIs:

- esp_wifi_set_bandwidth,
- esp_wifi_set_protocol,
- esp_wifi_set_config related
- esp_wifi_set_mode

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init

esp_err_t **esp_wifi_connect** (void)

Connect the ESP32 WiFi station to the AP.

Attention 1. This API only impact WIFI_MODE_STA or WIFI_MODE_APSTA mode

Attention 2. If the ESP32 is connected to an AP, call esp_wifi_disconnect to disconnect.

Attention 3. The scanning triggered by esp_wifi_start_scan() will not be effective until connection between ESP32 and the AP is established. If ESP32 is scanning and connecting at the same time, ESP32 will abort scanning and return a warning message and error number ESP_ERR_WIFI_STATE. If you want to do reconnection after ESP32 received disconnect event, remember to add the maximum retry time, otherwise the called scan will not work. This is especially true when the AP doesn't exist, and you still try reconnection after ESP32 received disconnect event with the reason code WIFI_REASON_NO_AP_FOUND.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_NOT_STARTED: WiFi is not started by esp_wifi_start
- ESP_ERR_WIFI_CONN: WiFi internal error, station or soft-AP control block wrong
- ESP_ERR_WIFI_SSID: SSID of AP which station connects is invalid

esp_err_t **esp_wifi_disconnect** (void)

Disconnect the ESP32 WiFi station from the AP.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi was not initialized by esp_wifi_init
- ESP_ERR_WIFI_NOT_STARTED: WiFi was not started by esp_wifi_start
- ESP_FAIL: other WiFi internal errors

esp_err_t **esp_wifi_clear_fast_connect** (void)

Currently this API is just an stub API.

Return

- ESP_OK: succeed
- others: fail

esp_err_t **esp_wifi_deauth_sta** (uint16_t aid)

deauthenticate all stations or associated id equals to aid

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_NOT_STARTED: WiFi was not started by esp_wifi_start
- ESP_ERR_INVALID_ARG: invalid argument
- ESP_ERR_WIFI_MODE: WiFi mode is wrong

Parameters

- `aid`: when `aid` is 0, deauthenticate all stations, otherwise deauthenticate station whose associated id is `aid`

esp_err_t **esp_wifi_scan_start** (`const` *wifi_scan_config_t* **config*, `bool` *block*)

Scan all available APs.

Attention If this API is called, the found APs are stored in WiFi driver dynamic allocated memory and the will be freed in `esp_wifi_scan_get_ap_records`, so generally, call `esp_wifi_scan_get_ap_records` to cause the memory to be freed once the scan is done

Attention The values of maximum active scan time and passive scan time per channel are limited to 1500 milliseconds. Values above 1500ms may cause station to disconnect from AP and are not recommended.

Return

- `ESP_OK`: succeed
- `ESP_ERR_WIFI_NOT_INIT`: WiFi is not initialized by `esp_wifi_init`
- `ESP_ERR_WIFI_NOT_STARTED`: WiFi was not started by `esp_wifi_start`
- `ESP_ERR_WIFI_TIMEOUT`: blocking scan is timeout
- `ESP_ERR_WIFI_STATE`: wifi still connecting when invoke `esp_wifi_scan_start`
- others: refer to error code in `esp_err.h`

Parameters

- `config`: configuration of scanning
- `block`: if `block` is true, this API will block the caller until the scan is done, otherwise it will return immediately

esp_err_t **esp_wifi_scan_stop** (`void`)

Stop the scan in process.

Return

- `ESP_OK`: succeed
- `ESP_ERR_WIFI_NOT_INIT`: WiFi is not initialized by `esp_wifi_init`
- `ESP_ERR_WIFI_NOT_STARTED`: WiFi is not started by `esp_wifi_start`

esp_err_t **esp_wifi_scan_get_ap_num** (`uint16_t` **number*)

Get number of APs found in last scan.

Attention This API can only be called when the scan is completed, otherwise it may get wrong value.

Return

- `ESP_OK`: succeed
- `ESP_ERR_WIFI_NOT_INIT`: WiFi is not initialized by `esp_wifi_init`
- `ESP_ERR_WIFI_NOT_STARTED`: WiFi is not started by `esp_wifi_start`
- `ESP_ERR_INVALID_ARG`: invalid argument

Parameters

- [out] `number`: store number of APIs found in last scan

esp_err_t **esp_wifi_scan_get_ap_records** (`uint16_t` **number*, *wifi_ap_record_t* **ap_records*)

Get AP list found in last scan.

Return

- `ESP_OK`: succeed
- `ESP_ERR_WIFI_NOT_INIT`: WiFi is not initialized by `esp_wifi_init`
- `ESP_ERR_WIFI_NOT_STARTED`: WiFi is not started by `esp_wifi_start`
- `ESP_ERR_INVALID_ARG`: invalid argument
- `ESP_ERR_NO_MEM`: out of memory

Parameters

- [inout] `number`: As input param, it stores max AP number `ap_records` can hold. As output param, it receives the actual AP number this API returns.
- `ap_records`: *wifi_ap_record_t* array to hold the found APs

esp_err_t **esp_wifi_sta_get_ap_info** (*wifi_ap_record_t* **ap_info*)

Get information of AP which the ESP32 station is associated with.

Attention When the obtained country information is empty, it means that the AP does not carry country information

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_CONN: The station interface don't initialized
- ESP_ERR_WIFI_NOT_CONNECT: The station is in disconnect status

Parameters

- ap_info: the *wifi_ap_record_t* to hold AP information sta can get the connected ap's phy mode info through the struct member phy_11b, phy_11g, phy_11n, phy_lr in the *wifi_ap_record_t* struct. For example, phy_11b = 1 imply that ap support 802.11b mode

esp_err_t **esp_wifi_set_ps** (*wifi_ps_type_t* type)

Set current WiFi power save type.

Attention Default power save type is WIFI_PS_MIN_MODEM.

Return ESP_OK: succeed

Parameters

- type: power save type

esp_err_t **esp_wifi_get_ps** (*wifi_ps_type_t* *type)

Get current WiFi power save type.

Attention Default power save type is WIFI_PS_MIN_MODEM.

Return ESP_OK: succeed

Parameters

- [out] type: store current power save type

esp_err_t **esp_wifi_set_protocol** (*wifi_interface_t* ifx, uint8_t protocol_bitmap)

Set protocol type of specified interface The default protocol is (WIFI_PROTOCOL_11B|WIFI_PROTOCOL_11G|WIFI_PROT

Attention Currently we only support 802.11b or 802.11g or 802.11bgn mode

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_IF: invalid interface
- others: refer to error codes in esp_err.h

Parameters

- ifx: interfaces
- protocol_bitmap: WiFi protocol bitmap

esp_err_t **esp_wifi_get_protocol** (*wifi_interface_t* ifx, uint8_t *protocol_bitmap)

Get the current protocol bitmap of the specified interface.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_IF: invalid interface
- ESP_ERR_INVALID_ARG: invalid argument
- others: refer to error codes in esp_err.h

Parameters

- ifx: interface
- [out] protocol_bitmap: store current WiFi protocol bitmap of interface ifx

esp_err_t **esp_wifi_set_bandwidth** (*wifi_interface_t* ifx, *wifi_bandwidth_t* bw)

Set the bandwidth of ESP32 specified interface.

Attention 1. API return false if try to configure an interface that is not enabled

Attention 2. WIFI_BW_HT40 is supported only when the interface support 11N

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_IF: invalid interface
- ESP_ERR_INVALID_ARG: invalid argument
- others: refer to error codes in esp_err.h

Parameters

- ifx: interface to be configured

- bw: bandwidth

`esp_err_t esp_wifi_get_bandwidth (wifi_interface_t ifx, wifi_bandwidth_t *bw)`

Get the bandwidth of ESP32 specified interface.

Attention 1. API return false if try to get a interface that is not enable

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_IF: invalid interface
- ESP_ERR_INVALID_ARG: invalid argument

Parameters

- ifx: interface to be configured
- [out] bw: store bandwidth of interface ifx

`esp_err_t esp_wifi_set_channel (uint8_t primary, wifi_second_chan_t second)`

Set primary/secondary channel of ESP32.

Attention 1. This API should be called after esp_wifi_start()

Attention 2. When ESP32 is in STA mode, this API should not be called when STA is scanning or connecting to an external AP

Attention 3. When ESP32 is in softAP mode, this API should not be called when softAP has connected to external STAs

Attention 4. When ESP32 is in STA+softAP mode, this API should not be called when in the scenarios described above

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_IF: invalid interface
- ESP_ERR_INVALID_ARG: invalid argument

Parameters

- primary: for HT20, primary is the channel number, for HT40, primary is the primary channel
- second: for HT20, second is ignored, for HT40, second is the second channel

`esp_err_t esp_wifi_get_channel (uint8_t *primary, wifi_second_chan_t *second)`

Get the primary/secondary channel of ESP32.

Attention 1. API return false if try to get a interface that is not enable

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument

Parameters

- primary: store current primary channel
- [out] second: store current second channel

`esp_err_t esp_wifi_set_country (const wifi_country_t *country)`

configure country info

Attention 1. The default country is {.cc=" CN" , .schan=1, .nchan=13, policy=WIFI_COUNTRY_POLICY_AUTO}

Attention 2. When the country policy is WIFI_COUNTRY_POLICY_AUTO, the country info of the AP to which the station is connected is used. E.g. if the configured country info is {.cc=" USA" , .schan=1, .nchan=11} and the country info of the AP to which the station is connected is {.cc=" JP" , .schan=1, .nchan=14} then the country info that will be used is {.cc=" JP" , .schan=1, .nchan=14}. If the station disconnected from the AP the country info is set back back to the country info of the station automatically, {.cc=" US" , .schan=1, .nchan=11} in the example.

Attention 3. When the country policy is WIFI_COUNTRY_POLICY_MANUAL, always use the configured country info.

Attention 4. When the country info is changed because of configuration or because the station connects to a different external AP, the country IE in probe response/beacon of the soft-AP is changed also.

Attention 5. The country configuration is not stored into flash

Attention 6. This API doesn't validate the per-country rules, it's up to the user to fill in all fields according to local regulations.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument

Parameters

- `country`: the configured country info

esp_err_t **esp_wifi_get_country** (*wifi_country_t* **country*)

get the current country info

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument

Parameters

- `country`: country info

esp_err_t **esp_wifi_set_mac** (*wifi_interface_t* *ifx*, **const** uint8_t *mac*[6])

Set MAC address of the ESP32 WiFi station or the soft-AP interface.

Attention 1. This API can only be called when the interface is disabled

Attention 2. ESP32 soft-AP and station have different MAC addresses, do not set them to be the same.

Attention 3. The bit 0 of the first byte of ESP32 MAC address can not be 1. For example, the MAC address can set to be "1a:XX:XX:XX:XX:XX", but can not be "15:XX:XX:XX:XX:XX".

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument
- ESP_ERR_WIFI_IF: invalid interface
- ESP_ERR_WIFI_MAC: invalid mac address
- ESP_ERR_WIFI_MODE: WiFi mode is wrong
- others: refer to error codes in esp_err.h

Parameters

- `ifx`: interface
- `mac`: the MAC address

esp_err_t **esp_wifi_get_mac** (*wifi_interface_t* *ifx*, uint8_t *mac*[6])

Get mac of specified interface.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument
- ESP_ERR_WIFI_IF: invalid interface

Parameters

- `ifx`: interface
- [out] `mac`: store mac of the interface ifx

esp_err_t **esp_wifi_set_promiscuous_rx_cb** (*wifi_promiscuous_cb_t* *cb*)

Register the RX callback function in the promiscuous mode.

Each time a packet is received, the registered callback function will be called.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init

Parameters

- `cb`: callback

esp_err_t **esp_wifi_set_promiscuous** (**bool** *en*)

Enable the promiscuous mode.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init

Parameters

- en: false - disable, true - enable

esp_err_t **esp_wifi_get_promiscuous** (bool *en)

Get the promiscuous mode.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument

Parameters

- [out] en: store the current status of promiscuous mode

esp_err_t **esp_wifi_set_promiscuous_filter** (const *wifi_promiscuous_filter_t* *filter)

Enable the promiscuous mode packet type filter.

Note The default filter is to filter all packets except WIFI_PKT_MISC

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init

Parameters

- filter: the packet type filtered in promiscuous mode.

esp_err_t **esp_wifi_get_promiscuous_filter** (*wifi_promiscuous_filter_t* *filter)

Get the promiscuous filter.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument

Parameters

- [out] filter: store the current status of promiscuous filter

esp_err_t **esp_wifi_set_promiscuous_ctrl_filter** (const *wifi_promiscuous_filter_t* *filter)

Enable subtype filter of the control packet in promiscuous mode.

Note The default filter is to filter none control packet.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init

Parameters

- filter: the subtype of the control packet filtered in promiscuous mode.

esp_err_t **esp_wifi_get_promiscuous_ctrl_filter** (*wifi_promiscuous_filter_t* *filter)

Get the subtype filter of the control packet in promiscuous mode.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_ARG: invalid argument

Parameters

- [out] filter: store the current status of subtype filter of the control packet in promiscuous mode

esp_err_t **esp_wifi_set_config** (*wifi_interface_t* interface, *wifi_config_t* *conf)

Set the configuration of the ESP32 STA or AP.

Attention 1. This API can be called only when specified interface is enabled, otherwise, API fail

Attention 2. For station configuration, bssid_set needs to be 0; and it needs to be 1 only when users need to check the MAC address of the AP.

Attention 3. ESP32 is limited to only one channel, so when in the soft-AP+station mode, the soft-AP will adjust its channel automatically to be the same as the channel of the ESP32 station.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument
- ESP_ERR_WIFI_IF: invalid interface
- ESP_ERR_WIFI_MODE: invalid mode
- ESP_ERR_WIFI_PASSWORD: invalid password
- ESP_ERR_WIFI_NVS: WiFi internal NVS error
- others: refer to the erro code in esp_err.h

Parameters

- interface: interface
- conf: station or soft-AP configuration

esp_err_t **esp_wifi_get_config** (*wifi_interface_t* interface, *wifi_config_t* *conf)

Get configuration of specified interface.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument
- ESP_ERR_WIFI_IF: invalid interface

Parameters

- interface: interface
- [out] conf: station or soft-AP configuration

esp_err_t **esp_wifi_ap_get_sta_list** (*wifi_sta_list_t* *sta)

Get STAs associated with soft-AP.

Attention SSC only API

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument
- ESP_ERR_WIFI_MODE: WiFi mode is wrong
- ESP_ERR_WIFI_CONN: WiFi internal error, the station/soft-AP control block is invalid

Parameters

- [out] sta: station list ap can get the connected sta' s phy mode info through the struct member phy_11b, phy_11g, phy_11n, phy_1r in the *wifi_sta_info_t* struct. For example, phy_11b = 1 imply that sta support 802.11b mode

esp_err_t **esp_wifi_ap_get_sta_aid** (const uint8_t mac[6], uint16_t *aid)

Get AID of STA connected with soft-AP.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument
- ESP_ERR_NOT_FOUND: Requested resource not found
- ESP_ERR_WIFI_MODE: WiFi mode is wrong
- ESP_ERR_WIFI_CONN: WiFi internal error, the station/soft-AP control block is invalid

Parameters

- mac: STA' s mac address
- [out] aid: Store the AID corresponding to STA mac

esp_err_t **esp_wifi_set_storage** (*wifi_storage_t* storage)

Set the WiFi API configuration storage type.

Attention 1. The default value is WIFI_STORAGE_FLASH

Return

- ESP_OK: succeed

- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument

Parameters

- storage: : storage type

esp_err_t **esp_wifi_set_vendor_ie** (bool *enable*, *wifi_vendor_ie_type_t* *type*, *wifi_vendor_ie_id_t* *idx*, const void **vnd_ie*)

Set 802.11 Vendor-Specific Information Element.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init()
- ESP_ERR_INVALID_ARG: Invalid argument, including if first byte of *vnd_ie* is not WIFI_VENDOR_IE_ELEMENT_ID (0xDD) or second byte is an invalid length.
- ESP_ERR_NO_MEM: Out of memory

Parameters

- *enable*: If true, specified IE is enabled. If false, specified IE is removed.
- *type*: Information Element type. Determines the frame type to associate with the IE.
- *idx*: Index to set or clear. Each IE type can be associated with up to two elements (indices 0 & 1).
- *vnd_ie*: Pointer to vendor specific element data. First 6 bytes should be a header with fields matching *wifi_vendor_ie_data_t*. If *enable* is false, this argument is ignored and can be NULL. Data does not need to remain valid after the function returns.

esp_err_t **esp_wifi_set_vendor_ie_cb** (*esp_vendor_ie_cb_t* *cb*, void **ctx*)

Register Vendor-Specific Information Element monitoring callback.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init

Parameters

- *cb*: Callback function
- *ctx*: Context argument, passed to callback function.

esp_err_t **esp_wifi_set_max_tx_power** (int8_t *power*)

Set maximum transmitting power after WiFi start.

Attention 1. Maximum power before wifi startup is limited by PHY init data bin.

Attention 2. The value set by this API will be mapped to the *max_tx_power* of the structure *wifi_country_t* variable.

Attention 3. Mapping Table {Power, max_tx_power} = {{8, 2}, {20, 5}, {28, 7}, {34, 8}, {44, 11}, {52, 13}, {56, 14}, {60, 15}, {66, 16}, {72, 18}, {80, 20}}.

Attention 4. Param power unit is 0.25dBm, range is [8, 84] corresponding to 2dBm - 20dBm.

Attention 5. Relationship between set value and actual value. As follows: {set value range, actual value} = {{[8, 19],8}, {[20, 27],20}, {[28, 33],28}, {[34, 43],34}, {[44, 51],44}, {[52, 55],52}, {[56, 59],56}, {[60, 65],60}, {[66, 71],66}, {[72, 79],72}, {[80, 84],80}}.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_NOT_START: WiFi is not started by esp_wifi_start
- ESP_ERR_WIFI_ARG: invalid argument, e.g. parameter is out of range

Parameters

- *power*: Maximum WiFi transmitting power.

esp_err_t **esp_wifi_get_max_tx_power** (int8_t **power*)

Get maximum transmitting power after WiFi start.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_NOT_START: WiFi is not started by esp_wifi_start
- ESP_ERR_WIFI_ARG: invalid argument

Parameters

- `power`: Maximum WiFi transmitting power, unit is 0.25dBm.

esp_err_t **esp_wifi_set_event_mask** (uint32_t *mask*)

Set mask to enable or disable some WiFi events.

Attention 1. Mask can be created by logical OR of various `WIFI_EVENT_MASK_` constants. Events which have corresponding bit set in the mask will not be delivered to the system event handler.

Attention 2. Default WiFi event mask is `WIFI_EVENT_MASK_AP_PROBEREQRCVED`.

Attention 3. There may be lots of stations sending probe request data around. Don't unmask this event unless you need to receive probe request data.

Return

- `ESP_OK`: succeed
- `ESP_ERR_WIFI_NOT_INIT`: WiFi is not initialized by `esp_wifi_init`

Parameters

- `mask`: WiFi event mask.

esp_err_t **esp_wifi_get_event_mask** (uint32_t **mask*)

Get mask of WiFi events.

Return

- `ESP_OK`: succeed
- `ESP_ERR_WIFI_NOT_INIT`: WiFi is not initialized by `esp_wifi_init`
- `ESP_ERR_WIFI_ARG`: invalid argument

Parameters

- `mask`: WiFi event mask.

esp_err_t **esp_wifi_80211_tx** (*wifi_interface_t* *ifx*, const void **buffer*, int *len*, bool *en_sys_seq*)

Send raw ieee80211 data.

Attention Currently only support for sending beacon/probe request/probe response/action and non-QoS data frame

Return

- `ESP_OK`: success
- `ESP_ERR_WIFI_IF`: Invalid interface
- `ESP_ERR_INVALID_ARG`: Invalid parameter
- `ESP_ERR_WIFI_NO_MEM`: out of memory

Parameters

- `ifx`: interface if the Wi-Fi mode is Station, the `ifx` should be `WIFI_IF_STA`. If the Wi-Fi mode is SoftAP, the `ifx` should be `WIFI_IF_AP`. If the Wi-Fi mode is Station+SoftAP, the `ifx` should be `WIFI_IF_STA` or `WIFI_IF_AP`. If the `ifx` is wrong, the API returns `ESP_ERR_WIFI_IF`.
- `buffer`: raw ieee80211 buffer
- `len`: the length of raw buffer, the `len` must be `<= 1500` Bytes and `>= 24` Bytes
- `en_sys_seq`: indicate whether use the internal sequence number. If `en_sys_seq` is false, the sequence in raw buffer is unchanged, otherwise it will be overwritten by WiFi driver with the system sequence number. Generally, if `esp_wifi_80211_tx` is called before the Wi-Fi connection has been set up, both `en_sys_seq==true` and `en_sys_seq==false` are fine. However, if the API is called after the Wi-Fi connection has been set up, `en_sys_seq` must be true, otherwise `ESP_ERR_WIFI_ARG` is returned.

esp_err_t **esp_wifi_set_csi_rx_cb** (*wifi_csi_cb_t* *cb*, void **ctx*)

Register the RX callback function of CSI data.

Each time a CSI data is received, the callback function will be called.

Return

- `ESP_OK`: succeed
- `ESP_ERR_WIFI_NOT_INIT`: WiFi is not initialized by `esp_wifi_init`

Parameters

- `cb`: callback
- `ctx`: context argument, passed to callback function

esp_err_t **esp_wifi_set_csi_config** (const *wifi_csi_config_t* **config*)

Set CSI data configuration.

return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_NOT_START: WiFi is not started by esp_wifi_start or promiscuous mode is not enabled
- ESP_ERR_INVALID_ARG: invalid argument

Parameters

- config: configuration

esp_err_t **esp_wifi_set_csi** (bool *en*)

Enable or disable CSI.

return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_NOT_START: WiFi is not started by esp_wifi_start or promiscuous mode is not enabled
- ESP_ERR_INVALID_ARG: invalid argument

Parameters

- en: true - enable, false - disable

esp_err_t **esp_wifi_set_ant_gpio** (const *wifi_ant_gpio_config_t* **config*)

Set antenna GPIO configuration.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_ARG: Invalid argument, e.g. parameter is NULL, invalid GPIO number etc

Parameters

- config: Antenna GPIO configuration.

esp_err_t **esp_wifi_get_ant_gpio** (*wifi_ant_gpio_config_t* **config*)

Get current antenna GPIO configuration.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_ARG: invalid argument, e.g. parameter is NULL

Parameters

- config: Antenna GPIO configuration.

esp_err_t **esp_wifi_set_ant** (const *wifi_ant_config_t* **config*)

Set antenna configuration.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_ARG: Invalid argument, e.g. parameter is NULL, invalid antenna mode or invalid GPIO number

Parameters

- config: Antenna configuration.

esp_err_t **esp_wifi_get_ant** (*wifi_ant_config_t* **config*)

Get current antenna configuration.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_ARG: invalid argument, e.g. parameter is NULL

Parameters

- config: Antenna configuration.

esp_err_t **esp_wifi_set_inactive_time** (*wifi_interface_t* ifx, uint16_t sec)

Set the inactive time of the ESP32 STA or AP.

Attention 1. For Station, If the station does not receive a beacon frame from the connected SoftAP during the inactive time, disconnect from SoftAP. Default 6s.

Attention 2. For SoftAP, If the softAP doesn't receive any data from the connected STA during inactive time, the softAP will force death the STA. Default is 300s.

Attention 3. The inactive time configuration is not stored into flash

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_NOT_STARTED: WiFi is not started by esp_wifi_start
- ESP_ERR_WIFI_ARG: invalid argument, For Station, if sec is less than 3. For SoftAP, if sec is less than 10.

Parameters

- ifx: interface to be configured.
- sec: Inactive time. Unit seconds.

esp_err_t **esp_wifi_get_inactive_time** (*wifi_interface_t* ifx, uint16_t *sec)

Get inactive time of specified interface.

Return

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_WIFI_ARG: invalid argument

Parameters

- ifx: Interface to be configured.
- sec: Inactive time. Unit seconds.

esp_err_t **esp_wifi_stats_dump** (uint32_t modules)

Dump WiFi statistics.

Return

- ESP_OK: succeed
- others: failed

Parameters

- modules: statistic modules to be dumped

Structures

struct **wifi_init_config_t**

WiFi stack configuration parameters passed to esp_wifi_init call.

Public Members

system_event_handler_t **event_handler**

WiFi event handler

wifi_osi_funcs_t ***osi_funcs**

WiFi OS functions

wpa_crypto_funcs_t **wpa_crypto_funcs**

WiFi station crypto functions when connect

int **static_rx_buf_num**

WiFi static RX buffer number

int **dynamic_rx_buf_num**

WiFi dynamic RX buffer number

int **tx_buf_type**

WiFi TX buffer type

int static_tx_buf_num
WiFi static TX buffer number

int dynamic_tx_buf_num
WiFi dynamic TX buffer number

int csi_enable
WiFi channel state information enable flag

int ampdu_rx_enable
WiFi AMPDU RX feature enable flag

int ampdu_tx_enable
WiFi AMPDU TX feature enable flag

int nvs_enable
WiFi NVS flash enable flag

int nano_enable
Nano option for printf/scan family enable flag

int tx_ba_win
WiFi Block Ack TX window size

int rx_ba_win
WiFi Block Ack RX window size

int wifi_task_core_id
WiFi Task Core ID

int beacon_max_len
WiFi softAP maximum length of the beacon

int mgmt_sbuf_num
WiFi management short buffer number, the minimum value is 6, the maximum value is 32

uint64_t feature_caps
Enables additional WiFi features and capabilities

int magic
WiFi init magic number, it should be the last field

Macros

ESP_ERR_WIFI_NOT_INIT
WiFi driver was not installed by esp_wifi_init

ESP_ERR_WIFI_NOT_STARTED
WiFi driver was not started by esp_wifi_start

ESP_ERR_WIFI_NOT_STOPPED
WiFi driver was not stopped by esp_wifi_stop

ESP_ERR_WIFI_IF
WiFi interface error

ESP_ERR_WIFI_MODE
WiFi mode error

ESP_ERR_WIFI_STATE
WiFi internal state error

ESP_ERR_WIFI_CONN
WiFi internal control block of station or soft-AP error

ESP_ERR_WIFI_NVS
WiFi internal NVS module error

ESP_ERR_WIFI_MAC
MAC address is invalid

ESP_ERR_WIFI_SSID
SSID is invalid

ESP_ERR_WIFI_PASSWORD
Password is invalid

ESP_ERR_WIFI_TIMEOUT
Timeout error

ESP_ERR_WIFI_WAKE_FAIL
WiFi is in sleep state(RF closed) and wakeup fail

ESP_ERR_WIFI_WOULD_BLOCK
The caller would block

ESP_ERR_WIFI_NOT_CONNECT
Station still in disconnect status

ESP_ERR_WIFI_POST
Failed to post the event to WiFi task

ESP_ERR_WIFI_INIT_STATE
Invalid WiFi state when init/deinit is called

ESP_ERR_WIFI_STOP_STATE
Returned when WiFi is stopping

WIFI_STATIC_TX_BUFFER_NUM

WIFI_DYNAMIC_TX_BUFFER_NUM

WIFI_CSI_ENABLED

WIFI_AMPDU_RX_ENABLED

WIFI_AMPDU_TX_ENABLED

WIFI_NVS_ENABLED

WIFI_NANO_FORMAT_ENABLED

WIFI_INIT_CONFIG_MAGIC

WIFI_DEFAULT_TX_BA_WIN

WIFI_DEFAULT_RX_BA_WIN

WIFI_TASK_CORE_ID

WIFI_SOFTAP_BEACON_MAX_LEN

WIFI_MGMT_SBUF_NUM

CONFIG_FEATURE_WPA3_SAE_BIT

WIFI_INIT_CONFIG_DEFAULT()

Type Definitions

typedef void (***wifi_promiscuous_cb_t**)(void *buf, *wifi_promiscuous_pkt_type_t* type)

The RX callback function in the promiscuous mode. Each time a packet is received, the callback function will be called.

Parameters

- **buf**: Data received. Type of data in buffer (*wifi_promiscuous_pkt_t* or *wifi_pkt_rx_ctrl_t*) indicated by 'type' parameter.
- **type**: promiscuous packet type.

```
typedef void (*esp_vendor_ie_cb_t) (void *ctx, wifi_vendor_ie_type_t type, const uint8_t sa[6],  
                                     const vendor_ie_data_t *vnd_ie, int rssi)
```

Function signature for received Vendor-Specific Information Element callback.

Parameters

- `ctx`: Context argument, as passed to `esp_wifi_set_vendor_ie_cb()` when registering callback.
- `type`: Information element type, based on frame type received.
- `sa`: Source 802.11 address.
- `vnd_ie`: Pointer to the vendor specific element data received.
- `rssi`: Received signal strength indication.

```
typedef void (*wifi_csi_cb_t) (void *ctx, wifi_csi_info_t *data)
```

The RX callback function of Channel State Information(CSI) data.

Each time a CSI data is received, the callback function will be called.

Parameters

- `ctx`: context argument, passed to `esp_wifi_set_csi_rx_cb()` when registering callback function.
- `data`: CSI data received. The memory that it points to will be deallocated after callback function returns.

Header File

- [esp_wifi/include/esp_wifi_types.h](#)

Unions

```
union wifi_config_t
```

```
#include <esp_wifi_types.h> Configuration data for ESP32 AP or STA.
```

The usage of this union (for ap or sta configuration) is determined by the accompanying interface argument passed to `esp_wifi_set_config()` or `esp_wifi_get_config()`

Public Members

```
wifi_ap_config_t ap  
configuration of AP
```

```
wifi_sta_config_t sta  
configuration of STA
```

Structures

```
struct wifi_country_t
```

Structure describing WiFi country-based regional restrictions.

Public Members

```
char cc[3]  
country code string
```

```
uint8_t schan  
start channel
```

```
uint8_t nchan  
total channel number
```

```
int8_t max_tx_power  
This field is used for getting WiFi maximum transmitting power, call esp_wifi_set_max_tx_power to set the maximum transmitting power.
```

wifi_country_policy_t **policy**

country policy

struct wifi_active_scan_time_t

Range of active scan times per channel.

Public Members

uint32_t min

minimum active scan time per channel, units: millisecond

uint32_t max

maximum active scan time per channel, units: millisecond, values above 1500ms may cause station to disconnect from AP and are not recommended.

struct wifi_scan_time_t

Aggregate of active & passive scan time per channel.

Public Members

wifi_active_scan_time_t **active**

active scan time per channel, units: millisecond.

uint32_t passive

passive scan time per channel, units: millisecond, values above 1500ms may cause station to disconnect from AP and are not recommended.

struct wifi_scan_config_t

Parameters for an SSID scan.

Public Members

uint8_t *ssid

SSID of AP

uint8_t *bssid

MAC address of AP

uint8_t channel

channel, scan the specific channel

bool show_hidden

enable to scan AP whose SSID is hidden

wifi_scan_type_t **scan_type**

scan type, active or passive

wifi_scan_time_t **scan_time**

scan time per channel

struct wifi_ap_record_t

Description of a WiFi AP.

Public Members

uint8_t bssid[6]

MAC address of AP

uint8_t ssid[33]

SSID of AP

uint8_t **primary**
channel of AP

wifi_second_chan_t **second**
secondary channel of AP

int8_t **rss**
signal strength of AP

wifi_auth_mode_t **authmode**
authmode of AP

wifi_cipher_type_t **pairwise_cipher**
pairwise cipher of AP

wifi_cipher_type_t **group_cipher**
group cipher of AP

wifi_ant_t **ant**
antenna used to receive beacon from AP

uint32_t **phy_11b** : 1
bit: 0 flag to identify if 11b mode is enabled or not

uint32_t **phy_11g** : 1
bit: 1 flag to identify if 11g mode is enabled or not

uint32_t **phy_11n** : 1
bit: 2 flag to identify if 11n mode is enabled or not

uint32_t **phy_1r** : 1
bit: 3 flag to identify if low rate is enabled or not

uint32_t **wps** : 1
bit: 4 flag to identify if WPS is supported or not

uint32_t **reserved** : 27
bit: 5..31 reserved

wifi_country_t **country**
country information of AP

struct wifi_scan_threshold_t
Structure describing parameters for a WiFi fast scan.

Public Members

int8_t **rss**
The minimum rssi to accept in the fast scan mode

wifi_auth_mode_t **authmode**
The weakest authmode to accept in the fast scan mode

struct wifi_pmf_config_t
Configuration structure for Protected Management Frame

Public Members

bool **capable**
Advertizes support for Protected Management Frame. Device will prefer to connect in PMF mode if other device also advertizes PMF capability.

bool **required**
Advertizes that Protected Management Frame is required. Device will not associate to non-PMF capable devices.

struct wifi_ap_config_t

Soft-AP configuration settings for the ESP32.

Public Members**uint8_t ssid[32]**

SSID of ESP32 soft-AP. If ssid_len field is 0, this must be a Null terminated string. Otherwise, length is set according to ssid_len.

uint8_t password[64]

Password of ESP32 soft-AP.

uint8_t ssid_len

Optional length of SSID field.

uint8_t channel

Channel of ESP32 soft-AP

wifi_auth_mode_t authmode

Auth mode of ESP32 soft-AP. Do not support AUTH_WEP in soft-AP mode

uint8_t ssid_hidden

Broadcast SSID or not, default 0, broadcast the SSID

uint8_t max_connection

Max number of stations allowed to connect in, default 4, max 10

uint16_t beacon_interval

Beacon interval which should be multiples of 100. Unit: TU(time unit, 1 TU = 1024 us). Range: 100 ~ 60000. Default value: 100

struct wifi_sta_config_t

STA configuration settings for the ESP32.

Public Members**uint8_t ssid[32]**

SSID of target AP.

uint8_t password[64]

Password of target AP.

wifi_scan_method_t scan_method

do all channel scan or fast scan

bool bssid_set

whether set MAC address of target AP or not. Generally, station_config.bssid_set needs to be 0; and it needs to be 1 only when users need to check the MAC address of the AP.

uint8_t bssid[6]

MAC address of target AP

uint8_t channel

channel of target AP. Set to 1~13 to scan starting from the specified channel before connecting to AP. If the channel of AP is unknown, set it to 0.

uint16_t listen_interval

Listen interval for ESP32 station to receive beacon when WIFI_PS_MAX_MODEM is set. Units: AP beacon intervals. Defaults to 3 if set to 0.

wifi_sort_method_t sort_method

sort the connect AP in the list by rssi or security mode

wifi_scan_threshold_t threshold

When `sort_method` is set, only APs which have an auth mode that is more secure than the selected auth mode and a signal stronger than the minimum RSSI will be used.

wifi_pmf_config_t pmf_cfg

Configuration for Protected Management Frame. Will be advertised in RSN Capabilities in RSN IE.

struct wifi_sta_info_t

Description of STA associated with AP.

Public Members

uint8_t **mac**[6]

mac address

int8_t **rss**

current average rssi of sta connected

uint32_t **phy_11b** : 1

bit: 0 flag to identify if 11b mode is enabled or not

uint32_t **phy_11g** : 1

bit: 1 flag to identify if 11g mode is enabled or not

uint32_t **phy_11n** : 1

bit: 2 flag to identify if 11n mode is enabled or not

uint32_t **phy_lr** : 1

bit: 3 flag to identify if low rate is enabled or not

uint32_t **reserved** : 28

bit: 4..31 reserved

struct wifi_sta_list_t

List of stations associated with the ESP32 Soft-AP.

Public Members

wifi_sta_info_t **sta**[ESP_WIFI_MAX_CONN_NUM]

station list

int **num**

number of stations in the list (other entries are invalid)

struct vendor_ie_data_t

Vendor Information Element header.

The first bytes of the Information Element will match this header. Payload follows.

Public Members

uint8_t **element_id**

Should be set to WIFI_VENDOR_IE_ELEMENT_ID (0xDD)

uint8_t **length**

Length of all bytes in the element data following this field. Minimum 4.

uint8_t **vendor_oui**[3]

Vendor identifier (OUI).

uint8_t **vendor_oui_type**

Vendor-specific OUI type.

`uint8_t payload[0]`

Payload. Length is equal to value in 'length' field, minus 4.

struct wifi_pkt_rx_ctrl_t

Received packet radio metadata header, this is the common header at the beginning of all promiscuous mode RX callback buffers.

Public Members

signed **rssi** : 8

Received Signal Strength Indicator(RSSI) of packet. unit: dBm

unsigned **rate** : 5

PHY rate encoding of the packet. Only valid for non HT(11bg) packet

unsigned **__pad0__** : 1

reserve

unsigned **sig_mode** : 2

0: non HT(11bg) packet; 1: HT(11n) packet; 3: VHT(11ac) packet

unsigned **__pad1__** : 16

reserve

unsigned **mcs** : 7

Modulation Coding Scheme. If is HT(11n) packet, shows the modulation, range from 0 to 76(MSC0 ~ MCS76)

unsigned **cwb** : 1

Channel Bandwidth of the packet. 0: 20MHz; 1: 40MHz

unsigned **__pad2__** : 16

reserve

unsigned **smoothing** : 1

reserve

unsigned **not_sounding** : 1

reserve

unsigned **__pad3__** : 1

reserve

unsigned **aggregation** : 1

Aggregation. 0: MPDU packet; 1: AMPDU packet

unsigned **stbc** : 2

Space Time Block Code(STBC). 0: non STBC packet; 1: STBC packet

unsigned **fec_coding** : 1

Flag is set for 11n packets which are LDPC

unsigned **sgi** : 1

Short Guide Interval(SGI). 0: Long GI; 1: Short GI

signed **noise_floor** : 8

noise floor of Radio Frequency Module(RF). unit: 0.25dBm

unsigned **ampdu_cnt** : 8

ampdu cnt

unsigned **channel** : 4

primary channel on which this packet is received

unsigned **secondary_channel** : 4

secondary channel on which this packet is received. 0: none; 1: above; 2: below

unsigned **__pad4__** : 8
reserve

unsigned **timestamp** : 32
timestamp. The local time when this packet is received. It is precise only if modem sleep or light sleep is not enabled. unit: microsecond

unsigned **__pad5__** : 32
reserve

unsigned **__pad6__** : 31
reserve

unsigned **ant** : 1
antenna number from which this packet is received. 0: WiFi antenna 0; 1: WiFi antenna 1

unsigned **sig_len** : 12
length of packet including Frame Check Sequence(FCS)

unsigned **__pad7__** : 12
reserve

unsigned **rx_state** : 8
state of the packet. 0: no error; others: error numbers which are not public

struct wifi_promiscuous_pkt_t
Payload passed to ‘buf’ parameter of promiscuous mode RX callback.

Public Members

wifi_pkt_rx_ctrl_t **rx_ctrl**
metadata header

uint8_t **payload**[0]
Data or management payload. Length of payload is described by rx_ctrl.sig_len. Type of content determined by packet type argument of callback.

struct wifi_promiscuous_filter_t
Mask for filtering different packet types in promiscuous mode.

Public Members

uint32_t **filter_mask**
OR of one or more filter values WIFI_PROMIS_FILTER_*

struct wifi_csi_config_t
Channel state information(CSI) configuration type.

Public Members

bool **lltf_en**
enable to receive legacy long training field(lltf) data. Default enabled

bool **htltf_en**
enable to receive HT long training field(htltf) data. Default enabled

bool **stbc_htltf2_en**
enable to receive space time block code HT long training field(stbc-htltf2) data. Default enabled

bool **ltf_merge_en**
enable to generate htltf data by averaging lltf and ht_ltf data when receiving HT packet. Otherwise, use ht_ltf data directly. Default enabled

bool **channel_filter_en**
enable to turn on channel filter to smooth adjacent sub-carrier. Disable it to keep independence of adjacent sub-carrier. Default enabled

bool **manu_scale**
manually scale the CSI data by left shifting or automatically scale the CSI data. If set true, please set the shift bits. false: automatically. true: manually. Default false

uint8_t **shift**
manually left shift bits of the scale of the CSI data. The range of the left shift bits is 0~15

struct wifi_csi_info_t
CSI data type.

Public Members

wifi_pkt_rx_ctrl_t rx_ctrl
received packet radio metadata header of the CSI data

uint8_t **mac**[6]
source MAC address of the CSI data

bool **first_word_invalid**
first four bytes of the CSI data is invalid or not

int8_t ***buf**
buffer of CSI data

uint16_t **len**
length of CSI data

struct wifi_ant_gpio_t
WiFi GPIO configuration for antenna selection.

Public Members

uint8_t **gpio_select** : 1
Whether this GPIO is connected to external antenna switch

uint8_t **gpio_num** : 7
The GPIO number that connects to external antenna switch

struct wifi_ant_gpio_config_t
WiFi GPIOs configuration for antenna selection.

Public Members

wifi_ant_gpio_t gpio_cfg[4]
The configurations of GPIOs that connect to external antenna switch

struct wifi_ant_config_t
WiFi antenna configuration.

Public Members

wifi_ant_mode_t rx_ant_mode
WiFi antenna mode for receiving

wifi_ant_t rx_ant_default
Default antenna mode for receiving, it' s ignored if rx_ant_mode is not WIFI_ANT_MODE_AUTO

wifi_ant_mode_t tx_ant_mode

WiFi antenna mode for transmission, it can be set to WIFI_ANT_MODE_AUTO only if rx_ant_mode is set to WIFI_ANT_MODE_AUTO

uint8_t enabled_ant0 : 4

Index (in antenna GPIO configuration) of enabled WIFI_ANT_MODE_ANT0

uint8_t enabled_ant1 : 4

Index (in antenna GPIO configuration) of enabled WIFI_ANT_MODE_ANT1

struct wifi_event_sta_scan_done_t

Argument structure for WIFI_EVENT_SCAN_DONE event

Public Members**uint32_t status**

status of scanning APs: 0 — success, 1 - failure

uint8_t number

number of scan results

uint8_t scan_id

scan sequence number, used for block scan

struct wifi_event_sta_connected_t

Argument structure for WIFI_EVENT_STA_CONNECTED event

Public Members**uint8_t ssid[32]**

SSID of connected AP

uint8_t ssid_len

SSID length of connected AP

uint8_t bssid[6]

BSSID of connected AP

uint8_t channel

channel of connected AP

wifi_auth_mode_t authmode

authentication mode used by AP

struct wifi_event_sta_disconnected_t

Argument structure for WIFI_EVENT_STA_DISCONNECTED event

Public Members**uint8_t ssid[32]**

SSID of disconnected AP

uint8_t ssid_len

SSID length of disconnected AP

uint8_t bssid[6]

BSSID of disconnected AP

uint8_t reason

reason of disconnection

struct wifi_event_sta_authmode_change_t

Argument structure for WIFI_EVENT_STA_AUTHMODE_CHANGE event

Public Members

wifi_auth_mode_t **old_mode**
the old auth mode of AP

wifi_auth_mode_t **new_mode**
the new auth mode of AP

struct wifi_event_sta_wps_er_pin_t
Argument structure for WIFI_EVENT_STA_WPS_ER_PIN event

Public Members

uint8_t **pin_code**[8]
PIN code of station in enrollee mode

struct wifi_event_sta_wps_er_success_t
Argument structure for WIFI_EVENT_STA_WPS_ER_SUCCESS event

Public Members

uint8_t **ap_cred_cnt**
Number of AP credentials received

uint8_t **ssid**[MAX_SSID_LEN]
SSID of AP

uint8_t **passphrase**[MAX_PASSPHRASE_LEN]
Passphrase for the AP

struct *wifi_event_sta_wps_er_success_t*::[anonymous] **ap_cred**[MAX_WPS_AP_CRED]
All AP credentials received from WPS handshake

struct wifi_event_ap_staconnected_t
Argument structure for WIFI_EVENT_AP_STACONNECTED event

Public Members

uint8_t **mac**[6]
MAC address of the station connected to ESP32 soft-AP

uint8_t **aid**
the aid that ESP32 soft-AP gives to the station connected to

struct wifi_event_ap_stadisconnected_t
Argument structure for WIFI_EVENT_AP_STADISCONNECTED event

Public Members

uint8_t **mac**[6]
MAC address of the station disconnects to ESP32 soft-AP

uint8_t **aid**
the aid that ESP32 soft-AP gave to the station disconnects to

struct wifi_event_ap_probe_req_rx_t
Argument structure for WIFI_EVENT_AP_PROBEREQRCVED event

Public Members

int **rss**i
Received probe request signal strength

uint8_t **mac**[6]
MAC address of the station which send probe request

Macros

WIFI_IF_STA

WIFI_IF_AP

WIFI_PROTOCOL_11B

WIFI_PROTOCOL_11G

WIFI_PROTOCOL_11N

WIFI_PROTOCOL_LR

ESP_WIFI_MAX_CONN_NUM
max number of stations which can connect to ESP32 soft-AP

WIFI_VENDOR_IE_ELEMENT_ID

WIFI_PROMIS_FILTER_MASK_ALL
filter all packets

WIFI_PROMIS_FILTER_MASK_MGMT
filter the packets with type of WIFI_PKT_MGMT

WIFI_PROMIS_FILTER_MASK_CTRL
filter the packets with type of WIFI_PKT_CTRL

WIFI_PROMIS_FILTER_MASK_DATA
filter the packets with type of WIFI_PKT_DATA

WIFI_PROMIS_FILTER_MASK_MISC
filter the packets with type of WIFI_PKT_MISC

WIFI_PROMIS_FILTER_MASK_DATA_MPDU
filter the MPDU which is a kind of WIFI_PKT_DATA

WIFI_PROMIS_FILTER_MASK_DATA_AMPDU
filter the AMPDU which is a kind of WIFI_PKT_DATA

WIFI_PROMIS_FILTER_MASK_FCSFAIL
filter the FCS failed packets, do not open it in general

WIFI_PROMIS_CTRL_FILTER_MASK_ALL
filter all control packets

WIFI_PROMIS_CTRL_FILTER_MASK_WRAPPER
filter the control packets with subtype of Control Wrapper

WIFI_PROMIS_CTRL_FILTER_MASK_BAR
filter the control packets with subtype of Block Ack Request

WIFI_PROMIS_CTRL_FILTER_MASK_BA
filter the control packets with subtype of Block Ack

WIFI_PROMIS_CTRL_FILTER_MASK_PSPOLL
filter the control packets with subtype of PS-Poll

WIFI_PROMIS_CTRL_FILTER_MASK_RTS
filter the control packets with subtype of RTS

WIFI_PROMIS_CTRL_FILTER_MASK_CTS
filter the control packets with subtype of CTS

WIFI_PROMIS_CTRL_FILTER_MASK_ACK
filter the control packets with subtype of ACK

WIFI_PROMIS_CTRL_FILTER_MASK_CFEND
filter the control packets with subtype of CF-END

WIFI_PROMIS_CTRL_FILTER_MASK_CFENDACK
filter the control packets with subtype of CF-END+CF-ACK

WIFI_EVENT_MASK_ALL
mask all WiFi events

WIFI_EVENT_MASK_NONE
mask none of the WiFi events

WIFI_EVENT_MASK_AP_PROBEREQRECVD
mask SYSTEM_EVENT_AP_PROBEREQRECVD event

MAX_SSID_LEN

MAX_PASSPHRASE_LEN

MAX_WPS_AP_CRED

WIFI_STATIS_BUFFER

WIFI_STATIS_RXTX

WIFI_STATIS_HW

WIFI_STATIS_DIAG

WIFI_STATIS_ALL

Type Definitions

```
typedef esp_interface_t wifi_interface_t
```

Enumerations

```
enum wifi_mode_t
```

Values:

WIFI_MODE_NULL = 0
null mode

WIFI_MODE_STA
WiFi station mode

WIFI_MODE_AP
WiFi soft-AP mode

WIFI_MODE_APSTA
WiFi station + soft-AP mode

WIFI_MODE_MAX

```
enum wifi_country_policy_t
```

Values:

WIFI_COUNTRY_POLICY_AUTO
Country policy is auto, use the country info of AP to which the station is connected

WIFI_COUNTRY_POLICY_MANUAL
Country policy is manual, always use the configured country info

```
enum wifi_auth_mode_t
```

Values:

WIFI_AUTH_OPEN = 0
authenticate mode : open

WIFI_AUTH_WEP
authenticate mode : WEP

WIFI_AUTH_WPA_PSK
authenticate mode : WPA_PSK

WIFI_AUTH_WPA2_PSK
authenticate mode : WPA2_PSK

WIFI_AUTH_WPA_WPA2_PSK
authenticate mode : WPA_WPA2_PSK

WIFI_AUTH_WPA2_ENTERPRISE
authenticate mode : WPA2_ENTERPRISE

WIFI_AUTH_WPA3_PSK
authenticate mode : WPA3_PSK

WIFI_AUTH_WPA2_WPA3_PSK
authenticate mode : WPA2_WPA3_PSK

WIFI_AUTH_MAX

enum wifi_err_reason_t

Values:

WIFI_REASON_UNSPECIFIED = 1

WIFI_REASON_AUTH_EXPIRE = 2

WIFI_REASON_AUTH_LEAVE = 3

WIFI_REASON_ASSOC_EXPIRE = 4

WIFI_REASON_ASSOC_TOOMANY = 5

WIFI_REASON_NOT_AUTHED = 6

WIFI_REASON_NOT_ASSOCED = 7

WIFI_REASON_ASSOC_LEAVE = 8

WIFI_REASON_ASSOC_NOT_AUTHED = 9

WIFI_REASON_DISASSOC_PWRCAP_BAD = 10

WIFI_REASON_DISASSOC_SUPCHAN_BAD = 11

WIFI_REASON_IE_INVALID = 13

WIFI_REASON_MIC_FAILURE = 14

WIFI_REASON_4WAY_HANDSHAKE_TIMEOUT = 15

WIFI_REASON_GROUP_KEY_UPDATE_TIMEOUT = 16

WIFI_REASON_IE_IN_4WAY_DIFFERS = 17

WIFI_REASON_GROUP_CIPHER_INVALID = 18

WIFI_REASON_PAIRWISE_CIPHER_INVALID = 19

WIFI_REASON_AKMP_INVALID = 20

WIFI_REASON_UNSUPP_RSN_IE_VERSION = 21

WIFI_REASON_INVALID_RSN_IE_CAP = 22

WIFI_REASON_802_1X_AUTH_FAILED = 23

WIFI_REASON_CIPHER_SUITE_REJECTED = 24

WIFI_REASON_INVALID_PMKID = 53
WIFI_REASON_BEACON_TIMEOUT = 200
WIFI_REASON_NO_AP_FOUND = 201
WIFI_REASON_AUTH_FAIL = 202
WIFI_REASON_ASSOC_FAIL = 203
WIFI_REASON_HANDSHAKE_TIMEOUT = 204
WIFI_REASON_CONNECTION_FAIL = 205
WIFI_REASON_AP_TSF_RESET = 206

enum wifi_second_chan_t

Values:

WIFI_SECOND_CHAN_NONE = 0
the channel width is HT20
WIFI_SECOND_CHAN_ABOVE
the channel width is HT40 and the secondary channel is above the primary channel
WIFI_SECOND_CHAN_BELOW
the channel width is HT40 and the secondary channel is below the primary channel

enum wifi_scan_type_t

Values:

WIFI_SCAN_TYPE_ACTIVE = 0
active scan
WIFI_SCAN_TYPE_PASSIVE
passive scan

enum wifi_cipher_type_t

Values:

WIFI_CIPHER_TYPE_NONE = 0
the cipher type is none
WIFI_CIPHER_TYPE_WEP40
the cipher type is WEP40
WIFI_CIPHER_TYPE_WEP104
the cipher type is WEP104
WIFI_CIPHER_TYPE_TKIP
the cipher type is TKIP
WIFI_CIPHER_TYPE_CCMP
the cipher type is CCMP
WIFI_CIPHER_TYPE_TKIP_CCMP
the cipher type is TKIP and CCMP
WIFI_CIPHER_TYPE_AES_CMAC128
the cipher type is AES-CMAC-128
WIFI_CIPHER_TYPE_UNKNOWN
the cipher type is unknown

enum wifi_ant_t

WiFi antenna.

Values:

WIFI_ANT_ANT0
WiFi antenna 0

WIFI_ANT_ANT1

WiFi antenna 1

WIFI_ANT_MAX

Invalid WiFi antenna

enum wifi_scan_method_t

Values:

WIFI_FAST_SCAN = 0

Do fast scan, scan will end after find SSID match AP

WIFI_ALL_CHANNEL_SCAN

All channel scan, scan will end after scan all the channel

enum wifi_sort_method_t

Values:

WIFI_CONNECT_AP_BY_SIGNAL = 0

Sort match AP in scan list by RSSI

WIFI_CONNECT_AP_BY_SECURITY

Sort match AP in scan list by security mode

enum wifi_ps_type_t

Values:

WIFI_PS_NONE

No power save

WIFI_PS_MIN_MODEM

Minimum modem power saving. In this mode, station wakes up to receive beacon every DTIM period

WIFI_PS_MAX_MODEM

Maximum modem power saving. In this mode, interval to receive beacons is determined by the listen_interval parameter in [wifi_sta_config_t](#)

enum wifi_bandwidth_t

Values:

WIFI_BW_HT20 = 1

WIFI_BW_HT40

enum wifi_storage_t

Values:

WIFI_STORAGE_FLASH

all configuration will store in both memory and flash

WIFI_STORAGE_RAM

all configuration will only store in the memory

enum wifi_vendor_ie_type_t

Vendor Information Element type.

Determines the frame type that the IE will be associated with.

Values:

WIFI_VND_IE_TYPE_BEACON

WIFI_VND_IE_TYPE_PROBE_REQ

WIFI_VND_IE_TYPE_PROBE_RESP

WIFI_VND_IE_TYPE_ASSOC_REQ

WIFI_VND_IE_TYPE_ASSOC_RESP

enum wifi_vendor_ie_id_t

Vendor Information Element index.

Each IE type can have up to two associated vendor ID elements.

Values:

WIFI_VND_IE_ID_0

WIFI_VND_IE_ID_1

enum wifi_promiscuous_pkt_type_t

Promiscuous frame type.

Passed to promiscuous mode RX callback to indicate the type of parameter in the buffer.

Values:

WIFI_PKT_MGMT

Management frame, indicates 'buf' argument is *wifi_promiscuous_pkt_t*

WIFI_PKT_CTRL

Control frame, indicates 'buf' argument is *wifi_promiscuous_pkt_t*

WIFI_PKT_DATA

Data frame, indicates 'buf' argument is *wifi_promiscuous_pkt_t*

WIFI_PKT_MISC

Other type, such as MIMO etc. 'buf' argument is *wifi_promiscuous_pkt_t* but the payload is zero length.

enum wifi_ant_mode_t

WiFi antenna mode.

Values:

WIFI_ANT_MODE_ANT0

Enable WiFi antenna 0 only

WIFI_ANT_MODE_ANT1

Enable WiFi antenna 1 only

WIFI_ANT_MODE_AUTO

Enable WiFi antenna 0 and 1, automatically select an antenna

WIFI_ANT_MODE_MAX

Invalid WiFi enabled antenna

enum wifi_phy_rate_t

WiFi PHY rate encodings.

Values:

WIFI_PHY_RATE_1M_L = 0x00

1 Mbps with long preamble

WIFI_PHY_RATE_2M_L = 0x01

2 Mbps with long preamble

WIFI_PHY_RATE_5M_L = 0x02

5.5 Mbps with long preamble

WIFI_PHY_RATE_11M_L = 0x03

11 Mbps with long preamble

WIFI_PHY_RATE_2M_S = 0x05

2 Mbps with short preamble

WIFI_PHY_RATE_5M_S = 0x06

5.5 Mbps with short preamble

WIFI_PHY_RATE_11M_S = 0x07
11 Mbps with short preamble

WIFI_PHY_RATE_48M = 0x08
48 Mbps

WIFI_PHY_RATE_24M = 0x09
24 Mbps

WIFI_PHY_RATE_12M = 0x0A
12 Mbps

WIFI_PHY_RATE_6M = 0x0B
6 Mbps

WIFI_PHY_RATE_54M = 0x0C
54 Mbps

WIFI_PHY_RATE_36M = 0x0D
36 Mbps

WIFI_PHY_RATE_18M = 0x0E
18 Mbps

WIFI_PHY_RATE_9M = 0x0F
9 Mbps

WIFI_PHY_RATE_MCS0_LGI = 0x10
MCS0 with long GI, 6.5 Mbps for 20MHz, 13.5 Mbps for 40MHz

WIFI_PHY_RATE_MCS1_LGI = 0x11
MCS1 with long GI, 13 Mbps for 20MHz, 27 Mbps for 40MHz

WIFI_PHY_RATE_MCS2_LGI = 0x12
MCS2 with long GI, 19.5 Mbps for 20MHz, 40.5 Mbps for 40MHz

WIFI_PHY_RATE_MCS3_LGI = 0x13
MCS3 with long GI, 26 Mbps for 20MHz, 54 Mbps for 40MHz

WIFI_PHY_RATE_MCS4_LGI = 0x14
MCS4 with long GI, 39 Mbps for 20MHz, 81 Mbps for 40MHz

WIFI_PHY_RATE_MCS5_LGI = 0x15
MCS5 with long GI, 52 Mbps for 20MHz, 108 Mbps for 40MHz

WIFI_PHY_RATE_MCS6_LGI = 0x16
MCS6 with long GI, 58.5 Mbps for 20MHz, 121.5 Mbps for 40MHz

WIFI_PHY_RATE_MCS7_LGI = 0x17
MCS7 with long GI, 65 Mbps for 20MHz, 135 Mbps for 40MHz

WIFI_PHY_RATE_MCS0_SGI = 0x18
MCS0 with short GI, 7.2 Mbps for 20MHz, 15 Mbps for 40MHz

WIFI_PHY_RATE_MCS1_SGI = 0x19
MCS1 with short GI, 14.4 Mbps for 20MHz, 30 Mbps for 40MHz

WIFI_PHY_RATE_MCS2_SGI = 0x1A
MCS2 with short GI, 21.7 Mbps for 20MHz, 45 Mbps for 40MHz

WIFI_PHY_RATE_MCS3_SGI = 0x1B
MCS3 with short GI, 28.9 Mbps for 20MHz, 60 Mbps for 40MHz

WIFI_PHY_RATE_MCS4_SGI = 0x1C
MCS4 with short GI, 43.3 Mbps for 20MHz, 90 Mbps for 40MHz

WIFI_PHY_RATE_MCS5_SGI = 0x1D
MCS5 with short GI, 57.8 Mbps for 20MHz, 120 Mbps for 40MHz

WIFI_PHY_RATE_MCS6_SGI = 0x1E
MCS6 with short GI, 65 Mbps for 20MHz, 135 Mbps for 40MHz

WIFI_PHY_RATE_MCS7_SGI = 0x1F
MCS7 with short GI, 72.2 Mbps for 20MHz, 150 Mbps for 40MHz

WIFI_PHY_RATE_LORA_250K = 0x29
250 Kbps

WIFI_PHY_RATE_LORA_500K = 0x2A
500 Kbps

WIFI_PHY_RATE_MAX

enum wifi_event_t

WiFi event declarations

Values:

WIFI_EVENT_WIFI_READY = 0
ESP32 WiFi ready

WIFI_EVENT_SCAN_DONE
ESP32 finish scanning AP

WIFI_EVENT_STA_START
ESP32 station start

WIFI_EVENT_STA_STOP
ESP32 station stop

WIFI_EVENT_STA_CONNECTED
ESP32 station connected to AP

WIFI_EVENT_STA_DISCONNECTED
ESP32 station disconnected from AP

WIFI_EVENT_STA_AUTHMODE_CHANGE
the auth mode of AP connected by ESP32 station changed

WIFI_EVENT_STA_BEACON_TIMEOUT
ESP32 station beacon timeout

WIFI_EVENT_STA_WPS_ER_SUCCESS
ESP32 station wps succeeds in enrollee mode

WIFI_EVENT_STA_WPS_ER_FAILED
ESP32 station wps fails in enrollee mode

WIFI_EVENT_STA_WPS_ER_TIMEOUT
ESP32 station wps timeout in enrollee mode

WIFI_EVENT_STA_WPS_ER_PIN
ESP32 station wps pin code in enrollee mode

WIFI_EVENT_STA_WPS_ER_PBC_OVERLAP
ESP32 station wps overlap in enrollee mode

WIFI_EVENT_AP_START
ESP32 soft-AP start

WIFI_EVENT_AP_STOP
ESP32 soft-AP stop

WIFI_EVENT_AP_STACONNECTED
a station connected to ESP32 soft-AP

WIFI_EVENT_AP_STADISCONNECTED
a station disconnected from ESP32 soft-AP

WIFI_EVENT_AP_PROBEREQRCVED

Receive probe request packet in soft-AP interface

WIFI_EVENT_MAX

Invalid WiFi event ID

enum wifi_event_sta_wps_fail_reason_t

Argument structure for WIFI_EVENT_STA_WPS_ER_FAILED event

Values:

WPS_FAIL_REASON_NORMAL = 0

ESP32 WPS normal fail reason

WPS_FAIL_REASON_RECV_M2D

ESP32 WPS receive M2D frame

WPS_FAIL_REASON_MAX

SmartConfig

API Reference

Header File

- [esp_wifi/include/esp_smartconfig.h](#)

Functions

const char *esp_smartconfig_get_version (void)

Get the version of SmartConfig.

Return

- SmartConfig version const char.

esp_err_t esp_smartconfig_start (const *smartconfig_start_config_t* *config)

Start SmartConfig, config ESP device to connect AP. You need to broadcast information by phone APP. Device sniffer special packets from the air that containing SSID and password of target AP.

Attention 1. This API can be called in station or softAP-station mode.

Attention 2. Can not call esp_smartconfig_start twice before it finish, please call esp_smartconfig_stop first.

Return

- ESP_OK: succeed
- others: fail

Parameters

- config: pointer to smartconfig start configure structure

esp_err_t esp_smartconfig_stop (void)

Stop SmartConfig, free the buffer taken by esp_smartconfig_start.

Attention Whether connect to AP succeed or not, this API should be called to free memory taken by smartconfig_start.

Return

- ESP_OK: succeed
- others: fail

esp_err_t esp_esptouch_set_timeout (uint8_t time_s)

Set timeout of SmartConfig process.

Attention Timing starts from SC_STATUS_FIND_CHANNEL status. SmartConfig will restart if timeout.

Return

- ESP_OK: succeed
- others: fail

Parameters

- `time_s`: range 15s~255s, offset:45s.

esp_err_t **esp_smartconfig_set_type** (*smartconfig_type_t* type)

Set protocol type of SmartConfig.

Attention If users need to set the SmartConfig type, please set it before calling `esp_smartconfig_start`.

Return

- `ESP_OK`: succeed
- others: fail

Parameters

- type: Choose from the `smartconfig_type_t`.

esp_err_t **esp_smartconfig_fast_mode** (bool *enable*)

Set mode of SmartConfig. default normal mode.

Attention 1. Please call it before API `esp_smartconfig_start`.

Attention 2. Fast mode have corresponding APP(phone).

Attention 3. Two mode is compatible.

Return

- `ESP_OK`: succeed
- others: fail

Parameters

- `enable`: false-disable(default); true-enable;

Structures

struct smartconfig_event_got_ssid_pswd_t

Argument structure for `SC_EVENT_GOT_SSID_PSWD` event

Public Members

uint8_t **ssid**[32]

SSID of the AP. Null terminated string.

uint8_t **password**[64]

Password of the AP. Null terminated string.

bool **bssid_set**

whether set MAC address of target AP or not.

uint8_t **bssid**[6]

MAC address of target AP.

smartconfig_type_t **type**

Type of smartconfig(ESPTouch or AirKiss).

uint8_t **token**

Token from cellphone which is used to send ACK to cellphone.

uint8_t **cellphone_ip**[4]

IP address of cellphone.

struct smartconfig_start_config_t

Configure structure for `esp_smartconfig_start`

Public Members

bool **enable_log**

Enable smartconfig logs.

Macros

SMARTCONFIG_START_CONFIG_DEFAULT ()

Enumerations**enum smartconfig_type_t***Values:***SC_TYPE_ESPTOUCH = 0**

protocol: ESPTouch

SC_TYPE_AIRKISS

protocol: AirKiss

SC_TYPE_ESPTOUCH_AIRKISS

protocol: ESPTouch and AirKiss

enum smartconfig_event_t

Smartconfig event declarations

*Values:***SC_EVENT_SCAN_DONE**

ESP32 station smartconfig has finished to scan for APs

SC_EVENT_FOUND_CHANNEL

ESP32 station smartconfig has found the channel of the target AP

SC_EVENT_GOT_SSID_PSWD

ESP32 station smartconfig got the SSID and password

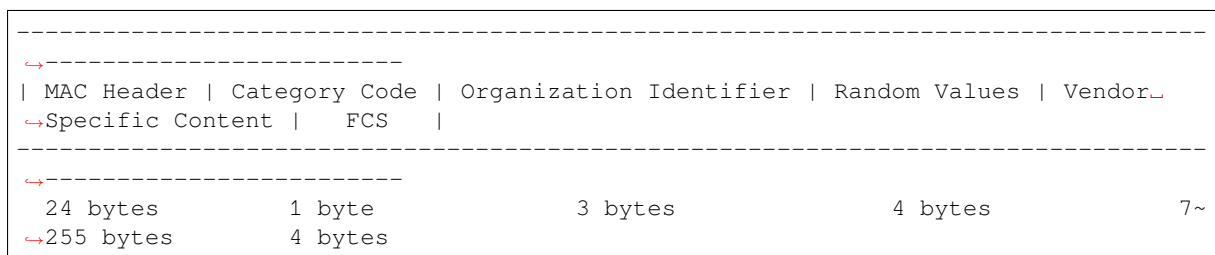
SC_EVENT_SEND_ACK_DONE

ESP32 station smartconfig has sent ACK to cellphone

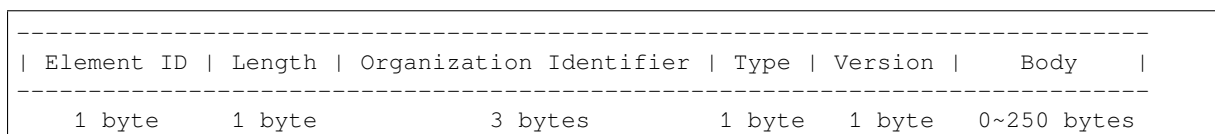
ESP-NOW

Overview ESP-NOW is a kind of connectionless Wi-Fi communication protocol that is defined by Espressif. In ESP-NOW, application data is encapsulated in a vendor-specific action frame and then transmitted from one Wi-Fi device to another without connection. CTR with CBC-MAC Protocol(CCMP) is used to protect the action frame for security. ESP-NOW is widely used in smart light, remote controlling, sensor, etc.

Frame Format ESP-NOW uses a vendor-specific action frame to transmit ESP-NOW data. The default ESP-NOW bit rate is 1 Mbps. The format of the vendor-specific action frame is as follows:



- **Category Code:** The Category Code field is set to the value(127) indicating the vendor-specific category.
- **Organization Identifier:** The Organization Identifier contains a unique identifier (0x18fe34), which is the first three bytes of MAC address applied by Espressif.
- **Random Value:** The Random Value field is used to prevent relay attacks.
- **Vendor Specific Content:** The Vendor Specific Content contains vendor-specific fields as follows:



- **Element ID:** The Element ID field is set to the value (221), indicating the vendor-specific element.
- **Length:** The length is the total length of Organization Identifier, Type, Version and Body.

- **Organization Identifier:** The Organization Identifier contains a unique identifier(0x18fe34), which is the first three bytes of MAC address applied by Espressif.
- **Type:** The Type field is set to the value (4) indicating ESP-NOW.
- **Version:** The Version field is set to the version of ESP-NOW.
- **Body:** The Body contains the ESP-NOW data.

As ESP-NOW is connectionless, the MAC header is a little different from that of standard frames. The FromDS and ToDS bits of FrameControl field are both 0. The first address field is set to the destination address. The second address field is set to the source address. The third address field is set to broadcast address (0xff:0xff:0xff:0xff:0xff).

Security

ESP-NOW uses the CCMP method, which is described in IEEE Std. 802.11-2012, to protect the vendor-specific action frame

- PMK is used to encrypt LMK with the AES-128 algorithm. Call `esp_now_set_pmk()` to set PMK. If PMK is not set, a default PMK will be used.
- LMK of the paired device is used to encrypt the vendor-specific action frame with the CCMP method. The maximum number of different LMKs is six. If the LMK of the paired device is not set, the vendor-specific action frame will not be encrypted.

Encrypting multicast vendor-specific action frame is not supported.

Initialization and De-initialization Call `esp_now_init()` to initialize ESP-NOW and `esp_now_deinit()` to de-initialize ESP-NOW. ESP-NOW data must be transmitted after Wi-Fi is started, so it is recommended to start Wi-Fi before initializing ESP-NOW and stop Wi-Fi after de-initializing ESP-NOW. When `esp_now_deinit()` is called, all of the information of paired devices will be deleted.

Add Paired Device Call `esp_now_add_peer()` to add the device to the paired device list before you send data to this device. The maximum number of paired devices is twenty. If security is enabled, the LMK must be set. You can send ESP-NOW data via both the Station and the SoftAP interface. Make sure that the interface is enabled before sending ESP-NOW data. A device with a broadcast MAC address must be added before sending broadcast data. The range of the channel of paired devices is from 0 to 14. If the channel is set to 0, data will be sent on the current channel. Otherwise, the channel must be set as the channel that the local device is on.

Send ESP-NOW Data Call `esp_now_send()` to send ESP-NOW data and `esp_now_register_send_cb` to register sending callback function. It will return `ESP_NOW_SEND_SUCCESS` in sending callback function if the data is received successfully on the MAC layer. Otherwise, it will return `ESP_NOW_SEND_FAIL`. Several reasons can lead to ESP-NOW fails to send data. For example, the destination device doesn't exist; the channels of the devices are not the same; the action frame is lost when transmitting on the air, etc. It is not guaranteed that application layer can receive the data. If necessary, send back ack data when receiving ESP-NOW data. If receiving ack data timeouts, retransmit the ESP-NOW data. A sequence number can also be assigned to ESP-NOW data to drop the duplicate data.

If there is a lot of ESP-NOW data to send, call `esp_now_send()` to send less than or equal to 250 bytes of data once a time. Note that too short interval between sending two ESP-NOW data may lead to disorder of sending callback function. So, it is recommended that sending the next ESP-NOW data after the sending callback function of the previous sending has returned. The sending callback function runs from a high-priority Wi-Fi task. So, do not do lengthy operations in the callback function. Instead, post the necessary data to a queue and handle it from a lower priority task.

Receiving ESP-NOW Data Call `esp_now_register_recv_cb` to register receiving callback function. Call the receiving callback function when receiving ESP-NOW. The receiving callback function also runs from the Wi-Fi task. So, do not do lengthy operations in the callback function. Instead, post the necessary data to a queue and handle it from a lower priority task.

API Reference

Header File

- `esp_wifi/include/esp_now.h`

Functions

esp_err_t **esp_now_init** (void)
Initialize ESPNOW function.

Return

- `ESP_OK` : succeed
- `ESP_ERR_ESPNOW_INTERNAL` : Internal error

esp_err_t **esp_now_deinit** (void)
De-initialize ESPNOW function.

Return

- `ESP_OK` : succeed

esp_err_t **esp_now_get_version** (uint32_t **version*)
Get the version of ESPNOW.

Return

- `ESP_OK` : succeed
- `ESP_ERR_ESPNOW_ARG` : invalid argument

Parameters

- *version*: ESPNOW version

esp_err_t **esp_now_register_recv_cb** (*esp_now_recv_cb_t* *cb*)
Register callback function of receiving ESPNOW data.

Return

- `ESP_OK` : succeed
- `ESP_ERR_ESPNOW_NOT_INIT` : ESPNOW is not initialized
- `ESP_ERR_ESPNOW_INTERNAL` : internal error

Parameters

- *cb*: callback function of receiving ESPNOW data

esp_err_t **esp_now_unregister_recv_cb** (void)
Unregister callback function of receiving ESPNOW data.

Return

- `ESP_OK` : succeed
- `ESP_ERR_ESPNOW_NOT_INIT` : ESPNOW is not initialized

esp_err_t **esp_now_register_send_cb** (*esp_now_send_cb_t* *cb*)
Register callback function of sending ESPNOW data.

Return

- `ESP_OK` : succeed
- `ESP_ERR_ESPNOW_NOT_INIT` : ESPNOW is not initialized
- `ESP_ERR_ESPNOW_INTERNAL` : internal error

Parameters

- *cb*: callback function of sending ESPNOW data

esp_err_t **esp_now_unregister_send_cb** (void)
Unregister callback function of sending ESPNOW data.

Return

- `ESP_OK` : succeed
- `ESP_ERR_ESPNOW_NOT_INIT` : ESPNOW is not initialized

esp_err_t **esp_now_send** (const uint8_t **peer_addr*, const uint8_t **data*, size_t *len*)
Send ESPNOW data.

Attention 1. If *peer_addr* is not NULL, send data to the peer whose MAC address matches *peer_addr*

Attention 2. If *peer_addr* is NULL, send data to all of the peers that are added to the peer list

Attention 3. The maximum length of data must be less than `ESP_NOW_MAX_DATA_LEN`

Attention 4. The buffer pointed to by data argument does not need to be valid after `esp_now_send` returns

Return

- `ESP_OK` : succeed
- `ESP_ERR_ESPNOW_NOT_INIT` : ESPNOW is not initialized
- `ESP_ERR_ESPNOW_ARG` : invalid argument
- `ESP_ERR_ESPNOW_INTERNAL` : internal error
- `ESP_ERR_ESPNOW_NO_MEM` : out of memory
- `ESP_ERR_ESPNOW_NOT_FOUND` : peer is not found
- `ESP_ERR_ESPNOW_IF` : current WiFi interface doesn't match that of peer

Parameters

- `peer_addr`: peer MAC address
- `data`: data to send
- `len`: length of data

esp_err_t `esp_now_add_peer` (**const** *esp_now_peer_info_t* *peer)

Add a peer to peer list.

Return

- `ESP_OK` : succeed
- `ESP_ERR_ESPNOW_NOT_INIT` : ESPNOW is not initialized
- `ESP_ERR_ESPNOW_ARG` : invalid argument
- `ESP_ERR_ESPNOW_FULL` : peer list is full
- `ESP_ERR_ESPNOW_NO_MEM` : out of memory
- `ESP_ERR_ESPNOW_EXIST` : peer has existed

Parameters

- `peer`: peer information

esp_err_t `esp_now_del_peer` (**const** `uint8_t` *peer_addr)

Delete a peer from peer list.

Return

- `ESP_OK` : succeed
- `ESP_ERR_ESPNOW_NOT_INIT` : ESPNOW is not initialized
- `ESP_ERR_ESPNOW_ARG` : invalid argument
- `ESP_ERR_ESPNOW_NOT_FOUND` : peer is not found

Parameters

- `peer_addr`: peer MAC address

esp_err_t `esp_now_mod_peer` (**const** *esp_now_peer_info_t* *peer)

Modify a peer.

Return

- `ESP_OK` : succeed
- `ESP_ERR_ESPNOW_NOT_INIT` : ESPNOW is not initialized
- `ESP_ERR_ESPNOW_ARG` : invalid argument
- `ESP_ERR_ESPNOW_FULL` : peer list is full

Parameters

- `peer`: peer information

esp_err_t `esp_now_get_peer` (**const** `uint8_t` *peer_addr, *esp_now_peer_info_t* *peer)

Get a peer whose MAC address matches `peer_addr` from peer list.

Return

- `ESP_OK` : succeed
- `ESP_ERR_ESPNOW_NOT_INIT` : ESPNOW is not initialized
- `ESP_ERR_ESPNOW_ARG` : invalid argument
- `ESP_ERR_ESPNOW_NOT_FOUND` : peer is not found

Parameters

- `peer_addr`: peer MAC address
- `peer`: peer information

esp_err_t **esp_now_fetch_peer** (bool *from_head*, *esp_now_peer_info_t* **peer*)

Fetch a peer from peer list. Only return the peer which address is unicast, for the multicast/broadcast address, the function will ignore and try to find the next in the peer list.

Return

- ESP_OK : succeed
- ESP_ERR_ESPNOW_NOT_INIT : ESPNOW is not initialized
- ESP_ERR_ESPNOW_ARG : invalid argument
- ESP_ERR_ESPNOW_NOT_FOUND : peer is not found

Parameters

- *from_head*: fetch from head of list or not
- *peer*: peer information

bool **esp_now_is_peer_exist** (const uint8_t **peer_addr*)

Peer exists or not.

Return

- true : peer exists
- false : peer not exists

Parameters

- *peer_addr*: peer MAC address

esp_err_t **esp_now_get_peer_num** (*esp_now_peer_num_t* **num*)

Get the number of peers.

Return

- ESP_OK : succeed
- ESP_ERR_ESPNOW_NOT_INIT : ESPNOW is not initialized
- ESP_ERR_ESPNOW_ARG : invalid argument

Parameters

- *num*: number of peers

esp_err_t **esp_now_set_pmk** (const uint8_t **pmk*)

Set the primary master key.

Attention 1. primary master key is used to encrypt local master key

Return

- ESP_OK : succeed
- ESP_ERR_ESPNOW_NOT_INIT : ESPNOW is not initialized
- ESP_ERR_ESPNOW_ARG : invalid argument

Parameters

- *pmk*: primary master key

Structures

struct esp_now_peer_info

ESPNOW peer information parameters.

Public Members

uint8_t **peer_addr**[ESP_NOW_ETH_ALEN]

ESPNOW peer MAC address that is also the MAC address of station or softap

uint8_t **lmk**[ESP_NOW_KEY_LEN]

ESPNOW peer local master key that is used to encrypt data

uint8_t **channel**

Wi-Fi channel that peer uses to send/receive ESPNOW data. If the value is 0, use the current channel which station or softap is on. Otherwise, it must be set as the channel that station or softap is on.

wifi_interface_t **ifidx**

Wi-Fi interface that peer uses to send/receive ESPNOW data

bool **encrypt**
ESPNow data that this peer sends/receives is encrypted or not

void ***priv**
ESPNow peer private data

struct esp_now_peer_num
Number of ESPNow peers which exist currently.

Public Members

int **total_num**
Total number of ESPNow peers, maximum value is ESP_NOW_MAX_TOTAL_PEER_NUM

int **encrypt_num**
Number of encrypted ESPNow peers, maximum value is ESP_NOW_MAX_ENCRYPT_PEER_NUM

Macros

ESP_ERR_ESPNow_BASE
ESPNow error number base.

ESP_ERR_ESPNow_NOT_INIT
ESPNow is not initialized.

ESP_ERR_ESPNow_ARG
Invalid argument

ESP_ERR_ESPNow_NO_MEM
Out of memory

ESP_ERR_ESPNow_FULL
ESPNow peer list is full

ESP_ERR_ESPNow_NOT_FOUND
ESPNow peer is not found

ESP_ERR_ESPNow_INTERNAL
Internal error

ESP_ERR_ESPNow_EXIST
ESPNow peer has existed

ESP_ERR_ESPNow_IF
Interface error

ESP_NOW_ETH_ALEN
Length of ESPNow peer MAC address

ESP_NOW_KEY_LEN
Length of ESPNow peer local master key

ESP_NOW_MAX_TOTAL_PEER_NUM
Maximum number of ESPNow total peers

ESP_NOW_MAX_ENCRYPT_PEER_NUM
Maximum number of ESPNow encrypted peers

ESP_NOW_MAX_DATA_LEN
Maximum length of ESPNow data which is sent very time

Type Definitions

typedef struct esp_now_peer_info esp_now_peer_info_t
ESPNow peer information parameters.

typedef struct esp_now_peer_num esp_now_peer_num_t
Number of ESPNow peers which exist currently.

```
typedef void (*esp_now_recv_cb_t) (const uint8_t *mac_addr, const uint8_t *data, int data_len)
```

Callback function of receiving ESPNOW data.

Parameters

- `mac_addr`: peer MAC address
- `data`: received data
- `data_len`: length of received data

```
typedef void (*esp_now_send_cb_t) (const uint8_t *mac_addr, esp_now_send_status_t status)
```

Callback function of sending ESPNOW data.

Parameters

- `mac_addr`: peer MAC address
- `status`: status of sending ESPNOW data (succeed or fail)

Enumerations

```
enum esp_now_send_status_t
```

Status of sending ESPNOW data .

Values:

```
ESP_NOW_SEND_SUCCESS = 0  
Send ESPNOW data successfully
```

```
ESP_NOW_SEND_FAIL  
Send ESPNOW data fail
```

ESP-MESH Programming Guide

This is a programming guide for ESP-MESH, including the API reference and coding examples. This guide is split into the following parts:

1. [ESP-MESH Programming Model](#)
2. [Writing an ESP-MESH Application](#)
3. [Self Organized Networking](#)
4. [Application Examples](#)
5. [API Reference](#)

For documentation regarding the ESP-MESH protocol, please see the [ESP-MESH API Guide](#). For more information about ESP-MESH Development Framework, please see [ESP-MESH Development Framework](#).

ESP-MESH Programming Model

Software Stack The ESP-MESH software stack is built atop the Wi-Fi Driver/FreeRTOS and may use the LwIP Stack in some instances (i.e. the root node). The following diagram illustrates the ESP-MESH software stack.

System Events An application interfaces with ESP-MESH via **ESP-MESH Events**. Since ESP-MESH is built atop the Wi-Fi stack, it is also possible for the application to interface with the Wi-Fi driver via the **Wi-Fi Event Task**. The following diagram illustrates the interfaces for the various System Events in an ESP-MESH application.

The `mesh_event_id_t` defines all possible ESP-MESH events and can indicate events such as the connection/disconnection of parent/child. Before ESP-MESH events can be used, the application must register a **Mesh Events handler** via `esp_event_handler_register()` to the default event task. The Mesh Events handler that is registered contain handlers for each ESP-MESH event relevant to the application.

Typical use cases of mesh events include using events such as `MESH_EVENT_PARENT_CONNECTED` and `MESH_EVENT_CHILD_CONNECTED` to indicate when a node can begin transmitting data upstream and downstream respectively. Likewise, `IP_EVENT_STA_GOT_IP` and `IP_EVENT_STA_LOST_IP` can be used to indicate when the root node can and cannot transmit data to the external IP network.

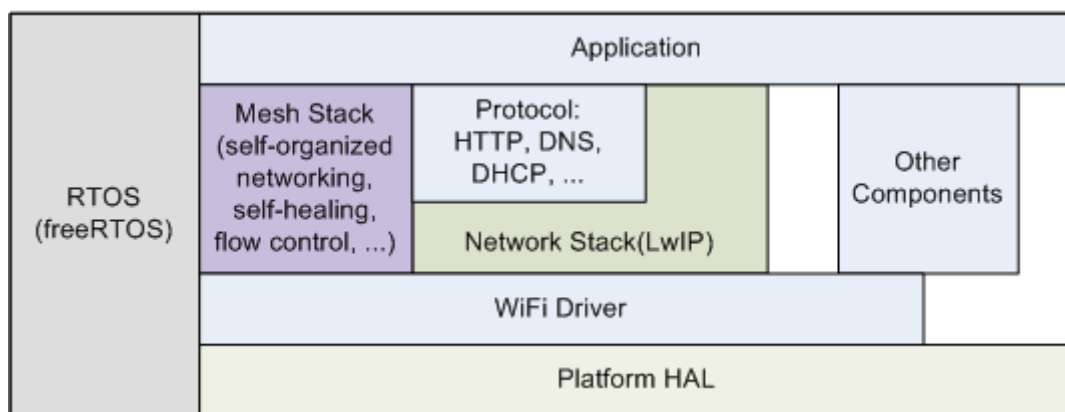


Fig. 2: ESP-MESH Software Stack

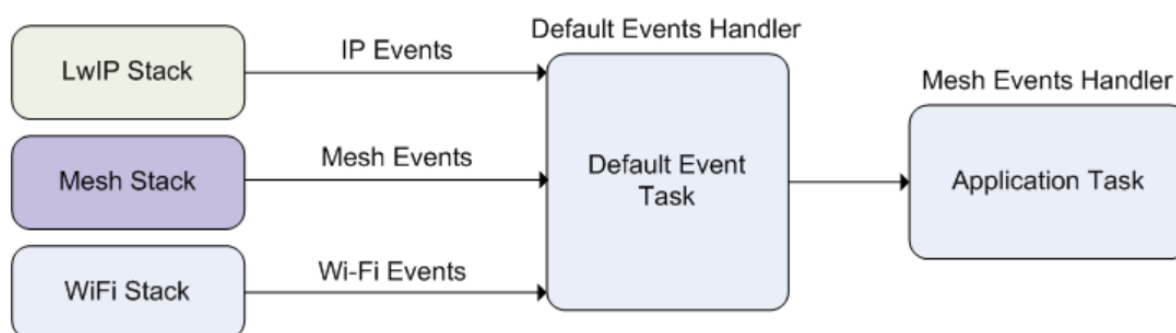


Fig. 3: ESP-MESH System Events Delivery

Warning: When using ESP-MESH under self-organized mode, users must ensure that no calls to Wi-Fi API are made. This is due to the fact that the self-organizing mode will internally make Wi-Fi API calls to connect/disconnect/scan etc. **Any Wi-Fi calls from the application (including calls from callbacks and handlers of Wi-Fi events) may interfere with ESP-MESH's self-organizing behavior.** Therefore, user's should not call Wi-Fi APIs after `esp_mesh_start()` is called, and before `esp_mesh_stop()` is called.

LwIP & ESP-MESH The application can access the ESP-MESH stack directly without having to go through the LwIP stack. The LwIP stack is only required by the root node to transmit/receive data to/from an external IP network. However, since every node can potentially become the root node (due to automatic root node selection), each node must still initialize the LwIP stack.

Each node is required to initialize LwIP by calling `tcpip_adapter_init()`. In order to prevent non-root node access to LwIP, the application should stop the following services after LwIP initialization:

- DHCP server service on the softAP interface.
- DHCP client service on the station interface.

The following code snippet demonstrates how to initialize LwIP for ESP-MESH applications.

```
/* tcpip initialization */
tcpip_adapter_init();
/*
 * for mesh
 * stop DHCP server on softAP interface by default
 * stop DHCP client on station interface by default
 */
```

(continues on next page)

(continued from previous page)

```
ESP_ERROR_CHECK(tcpip_adapter_dhcps_stop(TCPIP_ADAPTER_IF_AP));
ESP_ERROR_CHECK(tcpip_adapter_dhcpc_stop(TCPIP_ADAPTER_IF_STA));
```

Note: ESP-MESH requires a root node to be connected with a router. Therefore, in the event that a node becomes the root, **the corresponding handler must start the DHCP client service and immediately obtain an IP address.** Doing so will allow other nodes to begin transmitting/receiving packets to/from the external IP network. However, this step is unnecessary if static IP settings are used.

Writing an ESP-MESH Application The prerequisites for starting ESP-MESH is to initialize LwIP and Wi-Fi. The following code snippet demonstrates the necessary prerequisite steps before ESP-MESH itself can be initialized.

```
tcpip_adapter_init();
/*
 * for mesh
 * stop DHCP server on softAP interface by default
 * stop DHCP client on station interface by default
 */
ESP_ERROR_CHECK(tcpip_adapter_dhcps_stop(TCPIP_ADAPTER_IF_AP));
ESP_ERROR_CHECK(tcpip_adapter_dhcpc_stop(TCPIP_ADAPTER_IF_STA));

/* event initialization */
ESP_ERROR_CHECK(esp_event_loop_create_default());

/* Wi-Fi initialization */
wifi_init_config_t config = WIFI_INIT_CONFIG_DEFAULT();
ESP_ERROR_CHECK(esp_wifi_init(&config));
/* register IP events handler */
ESP_ERROR_CHECK(esp_event_handler_register(IP_EVENT, IP_EVENT_STA_GOT_IP, &ip_
↳event_handler, NULL));
ESP_ERROR_CHECK(esp_wifi_set_storage(WIFI_STORAGE_FLASH));
ESP_ERROR_CHECK(esp_wifi_start());
```

After initializing LwIP and Wi-Fi, the process of getting an ESP-MESH network up and running can be summarized into the following three steps:

1. *Initialize Mesh*
2. *Configuring an ESP-MESH Network*
3. *Start Mesh*

Initialize Mesh The following code snippet demonstrates how to initialize ESP-MESH

```
/* mesh initialization */
ESP_ERROR_CHECK(esp_mesh_init());
/* register mesh events handler */
ESP_ERROR_CHECK(esp_event_handler_register(MESH_EVENT, ESP_EVENT_ANY_ID, &mesh_
↳event_handler, NULL));
```

Configuring an ESP-MESH Network ESP-MESH is configured via `esp_mesh_set_config()` which receives its arguments using the `mesh_cfg_t` structure. The structure contains the following parameters used to configure ESP-MESH:

| Parameter | Description |
|------------------|---|
| Channel | Range from 1 to 14 |
| Mesh ID | ID of ESP-MESH Network, see mesh_addr_t |
| Router | Router Configuration, see mesh_router_t |
| Mesh AP | Mesh AP Configuration, see mesh_ap_cfg_t |
| Crypto Functions | Crypto Functions for Mesh IE, see mesh_crypto_funcs_t |

The following code snippet demonstrates how to configure ESP-MESH.

```

/* Enable the Mesh IE encryption by default */
mesh_cfg_t cfg = MESH_INIT_CONFIG_DEFAULT();
/* mesh ID */
memcpy((uint8_t *) &cfg.mesh_id, MESH_ID, 6);
/* channel (must match the router's channel) */
cfg.channel = CONFIG_MESH_CHANNEL;
/* router */
cfg.router.ssid_len = strlen(CONFIG_MESH_ROUTER_SSID);
memcpy((uint8_t *) &cfg.router.ssid, CONFIG_MESH_ROUTER_SSID, cfg.router.ssid_len);
memcpy((uint8_t *) &cfg.router.password, CONFIG_MESH_ROUTER_PASSWD,
        strlen(CONFIG_MESH_ROUTER_PASSWD));
/* mesh softAP */
cfg.mesh_ap.max_connection = CONFIG_MESH_AP_CONNECTIONS;
memcpy((uint8_t *) &cfg.mesh_ap.password, CONFIG_MESH_AP_PASSWD,
        strlen(CONFIG_MESH_AP_PASSWD));
ESP_ERROR_CHECK(esp_mesh_set_config(&cfg));

```

Start Mesh The following code snippet demonstrates how to start ESP-MESH.

```

/* mesh start */
ESP_ERROR_CHECK(esp_mesh_start());

```

After starting ESP-MESH, the application should check for ESP-MESH events to determine when it has connected to the network. After connecting, the application can start transmitting and receiving packets over the ESP-MESH network using [esp_mesh_send\(\)](#) and [esp_mesh_recv\(\)](#).

Self Organized Networking Self organized networking is a feature of ESP-MESH where nodes can autonomously scan/select/connect/reconnect to other nodes and routers. This feature allows an ESP-MESH network to operate with high degree of autonomy by making the network robust to dynamic network topologies and conditions. With self organized networking enabled, nodes in an ESP-MESH network are able to carry out the following actions without autonomously:

- Selection or election of the root node (see **Automatic Root Node Selection** in [Building a Network](#))
- Selection of a preferred parent node (see **Parent Node Selection** in [Building a Network](#))
- Automatic reconnection upon detecting a disconnection (see **Intermediate Parent Node Failure** in [Managing a Network](#))

When self organized networking is enabled, the ESP-MESH stack will internally make calls to Wi-Fi APIs. Therefore, **the application layer should not make any calls to Wi-Fi APIs whilst self organized networking is enabled as doing so would risk interfering with ESP-MESH.**

Toggle Self Organized Networking Self organized networking can be enabled or disabled by the application at runtime by calling the [esp_mesh_set_self_organized\(\)](#) function. The function has the two following parameters:

- `bool enable` specifies whether to enable or disable self organized networking.
- `bool select_parent` specifies whether a new parent node should be selected when enabling self organized networking. Selecting a new parent has different effects depending the node type and the node's current state. This parameter is unused when disabling self organized networking.

Disabling Self Organized Networking The following code snippet demonstrates how to disable self organized networking.

```
//Disable self organized networking
esp_mesh_set_self_organized(false, false);
```

ESP-MESH will attempt to maintain the node's current Wi-Fi state when disabling self organized networking.

- If the node was previously connected to other nodes, it will remain connected.
- If the node was previously disconnected and was scanning for a parent node or router, it will stop scanning.
- If the node was previously attempting to reconnect to a parent node or router, it will stop reconnecting.

Enabling Self Organized Networking ESP-MESH will attempt to maintain the node's current Wi-Fi state when enabling self organized networking. However, depending on the node type and whether a new parent is selected, the Wi-Fi state of the node can change. The following table shows effects of enabling self organized networking.

| Select Parent | Is Root Node | Effects |
|---------------|--------------|---|
| N | N | <ul style="list-style-type: none"> • Nodes already connected to a parent node will remain connected. • Nodes previously scanning for a parent nodes will stop scanning. Call <code>esp_mesh_connect()</code> to restart. |
| | Y | <ul style="list-style-type: none"> • A root node already connected to router will stay connected. • A root node disconnected from router will need to call <code>esp_mesh_connect()</code> to reconnect. |
| Y | N | <ul style="list-style-type: none"> • Nodes without a parent node will automatically select a preferred parent and connect. • Nodes already connected to a parent node will disconnect, reselect a preferred parent node, and connect. |
| | Y | <ul style="list-style-type: none"> • For a root node to connect to a parent node, it must give up its role as root. Therefore, a root node will disconnect from the router and all child nodes, select a preferred parent node, and connect. |

The following code snippet demonstrates how to enable self organized networking.

```
//Enable self organized networking and select a new parent
esp_mesh_set_self_organized(true, true);
```

(continues on next page)

```
...  
  
//Enable self organized networking and manually reconnect  
esp_mesh_set_self_organized(true, false);  
esp_mesh_connect();
```

Calling Wi-Fi API There can be instances in which an application may want to directly call Wi-Fi API whilst using ESP-MESH. For example, an application may want to manually scan for neighboring APs. However, **self organized networking must be disabled before the application calls any Wi-Fi APIs**. This will prevent the ESP-MESH stack from attempting to call any Wi-Fi APIs and potentially interfering with the application's calls.

Therefore, application calls to Wi-Fi APIs should be placed in between calls of `esp_mesh_set_self_organized()` which disable and enable self organized networking. The following code snippet demonstrates how an application can safely call `esp_wifi_scan_start()` whilst using ESP-MESH.

```
//Disable self organized networking  
esp_mesh_set_self_organized(0, 0);  
  
//Stop any scans already in progress  
esp_wifi_scan_stop();  
//Manually start scan. Will automatically stop when run to completion  
esp_wifi_scan_start();  
  
//Process scan results  
  
...  
  
//Re-enable self organized networking if still connected  
esp_mesh_set_self_organized(1, 0);  
  
...  
  
//Re-enable self organized networking if non-root and disconnected  
esp_mesh_set_self_organized(1, 1);  
  
...  
  
//Re-enable self organized networking if root and disconnected  
esp_mesh_set_self_organized(1, 0); //Don't select new parent  
esp_mesh_connect(); //Manually reconnect to router
```

Application Examples ESP-IDF contains these ESP-MESH example projects:

[The Internal Communication Example](#) demonstrates how to set up a ESP-MESH network and have the root node send a data packet to every node within the network.

[The Manual Networking Example](#) demonstrates how to use ESP-MESH without the self-organizing features. This example shows how to program a node to manually scan for a list of potential parent nodes and select a parent node based on custom criteria.

API Reference

Header File

- [esp_wifi/include/esp_mesh.h](#)

Functions

esp_err_t **esp_mesh_init** (void)

Mesh initialization.

- Check whether Wi-Fi is started.
- Initialize mesh global variables with default values.

Attention This API shall be called after Wi-Fi is started.

Return

- ESP_OK
- ESP_FAIL

esp_err_t **esp_mesh_deinit** (void)

Mesh de-initialization.

- Release resources and stop the mesh

Return

- ESP_OK
- ESP_FAIL

esp_err_t **esp_mesh_start** (void)

Start mesh.

- Initialize mesh IE.
- Start mesh network management service.
- Create TX and RX queues according to the configuration.
- Register mesh packets receive callback.

Attention This API shall be called after mesh initialization and configuration.

Return

- ESP_OK
- ESP_FAIL
- ESP_ERR_MESH_NOT_INIT
- ESP_ERR_MESH_NOT_CONFIG
- ESP_ERR_MESH_NO_MEMORY

esp_err_t **esp_mesh_stop** (void)

Stop mesh.

- Deinitialize mesh IE.
- Disconnect with current parent.
- Disassociate all currently associated children.
- Stop mesh network management service.
- Unregister mesh packets receive callback.
- Delete TX and RX queues.
- Release resources.
- Restore Wi-Fi softAP to default settings if Wi-Fi dual mode is enabled.

Return

- ESP_OK
- ESP_FAIL

esp_err_t **esp_mesh_send** (const *mesh_addr_t* *to, const *mesh_data_t* *data, int flag, const *mesh_opt_t* opt[], int opt_count)

Send a packet over the mesh network.

- Send a packet to any device in the mesh network.
- Send a packet to external IP network.

Attention This API is not reentrant.

Return

- ESP_OK
- ESP_FAIL
- ESP_ERR_MESH_ARGUMENT
- ESP_ERR_MESH_NOT_START

- ESP_ERR_MESH_DISCONNECTED
- ESP_ERR_MESH_OPT_UNKNOWN
- ESP_ERR_MESH_EXCEED_MTU
- ESP_ERR_MESH_NO_MEMORY
- ESP_ERR_MESH_TIMEOUT
- ESP_ERR_MESH_QUEUE_FULL
- ESP_ERR_MESH_NO_ROUTE_FOUND
- ESP_ERR_MESH_DISCARD

Parameters

- [in] `to`: the address of the final destination of the packet
 - If the packet is to the root, set this parameter to NULL.
 - If the packet is to an external IP network, set this parameter to the IPv4:PORT combination. This packet will be delivered to the root firstly, then the root will forward this packet to the final IP server address.
- [in] `data`: pointer to a sending mesh packet
 - Field size should not exceed MESH_MPS. Note that the size of one mesh packet should not exceed MESH_MTU.
 - Field proto should be set to data protocol in use (default is MESH_PROTO_BIN for binary).
 - Field tos should be set to transmission tos (type of service) in use (default is MESH_TOS_P2P for point-to-point reliable).
- [in] `flag`: bitmap for data sent
 - Speed up the route search
 - * If the packet is to the root and “to” parameter is NULL, set this parameter to 0.
 - * If the packet is to an internal device, MESH_DATA_P2P should be set.
 - * If the packet is to the root (“to” parameter isn’ t NULL) or to external IP network, MESH_DATA_TODS should be set.
 - * If the packet is from the root to an internal device, MESH_DATA_FROMDS should be set.
 - Specify whether this API is block or non-block, block by default
 - * If needs non-blocking, MESH_DATA_NONBLOCK should be set. Otherwise, may use `esp_mesh_send_block_time()` to specify a blocking time.
 - In the situation of the root change, MESH_DATA_DROP identifies this packet can be dropped by the new root for upstream data to external IP network, we try our best to avoid data loss caused by the root change, but there is a risk that the new root is running out of memory because most of memory is occupied by the pending data which isn’ t read out in time by `esp_mesh_rcv_toDS()`. Generally, we suggest `esp_mesh_rcv_toDS()` is called after a connection with IP network is created. Thus data outgoing to external IP network via socket is just from reading `esp_mesh_rcv_toDS()` which avoids unnecessary memory copy.
- [in] `opt`: options
 - In case of sending a packet to a certain group, MESH_OPT_SEND_GROUP is a good choice. In this option, the value field should be set to the target receiver addresses in this group.
 - Root sends a packet to an internal device, this packet is from external IP network in case the receiver device responds this packet, MESH_OPT_RECV_DS_ADDR is required to attach the target DS address.
- [in] `opt_count`: option count
 - Currently, this API only takes one option, so `opt_count` is only supported to be 1.

esp_err_t esp_mesh_send_block_time (uint32_t *time_ms*)

Set blocking time of `esp_mesh_send()`

Attention This API shall be called before mesh is started.

Return

- ESP_OK

Parameters

- [in] `time_ms`: blocking time of `esp_mesh_send()`, unit:ms

esp_err_t esp_mesh_rcv (*mesh_addr_t* **from*, *mesh_data_t* **data*, int *timeout_ms*, int **flag*, *mesh_opt_t* *opt*[], int *opt_count*)

Receive a packet targeted to self over the mesh network.

`flag` could be MESH_DATA_FROMDS or MESH_DATA_TODS.

Attention Mesh RX queue should be checked regularly to avoid running out of memory.

- Use `esp_mesh_get_rx_pending()` to check the number of packets available in the queue waiting to be received by applications.

Return

- `ESP_OK`
- `ESP_ERR_MESH_ARGUMENT`
- `ESP_ERR_MESH_NOT_START`
- `ESP_ERR_MESH_TIMEOUT`
- `ESP_ERR_MESH_DISCARD`

Parameters

- `[out] from`: the address of the original source of the packet
- `[out] data`: pointer to the received mesh packet
 - Field `proto` is the data protocol in use. Should follow it to parse the received data.
 - Field `tos` is the transmission tos (type of service) in use.
- `[in] timeout_ms`: wait time if a packet isn't immediately available (0:no wait, `port-MAX_DELAY`:wait forever)
- `[out] flag`: bitmap for data received
 - `MESH_DATA_FROMDS` represents data from external IP network
 - `MESH_DATA_TODS` represents data directed upward within the mesh network

Parameters

- `[out] opt`: options desired to receive
 - `MESH_OPT_RECV_DS_ADDR` attaches the DS address
- `[in] opt_count`: option count desired to receive
 - Currently, this API only takes one option, so `opt_count` is only supported to be 1.

`esp_err_t esp_mesh_recv_toDS(mesh_addr_t *from, mesh_addr_t *to, mesh_data_t *data, int timeout_ms, int *flag, mesh_opt_t opt[], int opt_count)`

Receive a packet targeted to external IP network.

- Root uses this API to receive packets destined to external IP network
- Root forwards the received packets to the final destination via socket.
- If no socket connection is ready to send out the received packets and this `esp_mesh_recv_toDS()` hasn't been called by applications, packets from the whole mesh network will be pending in toDS queue.

Use `esp_mesh_get_rx_pending()` to check the number of packets available in the queue waiting to be received by applications in case of running out of memory in the root.

Using `esp_mesh_set_xon_qsize()` users may configure the RX queue size, default:32. If this size is too large, and `esp_mesh_recv_toDS()` isn't called in time, there is a risk that a great deal of memory is occupied by the pending packets. If this size is too small, it will impact the efficiency on upstream. How to decide this value depends on the specific application scenarios.

`flag` could be `MESH_DATA_TODS`.

Attention This API is only called by the root.

Return

- `ESP_OK`
- `ESP_ERR_MESH_ARGUMENT`
- `ESP_ERR_MESH_NOT_START`
- `ESP_ERR_MESH_TIMEOUT`
- `ESP_ERR_MESH_DISCARD`
- `ESP_ERR_MESH_RECV_RELEASE`

Parameters

- `[out] from`: the address of the original source of the packet
- `[out] to`: the address contains remote IP address and port (IPv4:PORT)
- `[out] data`: pointer to the received packet
 - Contain the protocol and applications should follow it to parse the data.
- `[in] timeout_ms`: wait time if a packet isn't immediately available (0:no wait, `port-MAX_DELAY`:wait forever)
- `[out] flag`: bitmap for data received
 - `MESH_DATA_TODS` represents the received data target to external IP network. Root shall

forward this data to external IP network via the association with router.

Parameters

- [out] `opt`: options desired to receive
- [in] `opt_count`: option count desired to receive

esp_err_t **esp_mesh_set_config**(const *mesh_cfg_t* **config*)

Set mesh stack configuration.

- Use `MESH_INIT_CONFIG_DEFAULT()` to initialize the default values, mesh IE is encrypted by default.
- Mesh network is established on a fixed channel (1-14).
- Mesh event callback is mandatory.
- Mesh ID is an identifier of an MBSS. Nodes with the same mesh ID can communicate with each other.
- Regarding to the router configuration, if the router is hidden, BSSID field is mandatory.

If BSSID field isn't set and there exists more than one router with same SSID, there is a risk that more roots than one connected with different BSSID will appear. It means more than one mesh network is established with the same mesh ID.

Root conflict function could eliminate redundant roots connected with the same BSSID, but couldn't handle roots connected with different BSSID. Because users might have such requirements of setting up routers with same SSID for the future replacement. But in that case, if the above situations happen, please make sure applications implement forward functions on the root to guarantee devices in different mesh networks can communicate with each other. `max_connection` of mesh softAP is limited by the max number of Wi-Fi softAP supported (max:10).

Attention This API shall be called before mesh is started after mesh is initialized.

Return

- `ESP_OK`
- `ESP_ERR_MESH_ARGUMENT`
- `ESP_ERR_MESH_NOT_ALLOWED`

Parameters

- [in] `config`: pointer to mesh stack configuration

esp_err_t **esp_mesh_get_config**(*mesh_cfg_t* **config*)

Get mesh stack configuration.

Return

- `ESP_OK`
- `ESP_ERR_MESH_ARGUMENT`

Parameters

- [out] `config`: pointer to mesh stack configuration

esp_err_t **esp_mesh_set_router**(const *mesh_router_t* **router*)

Get router configuration.

Attention This API is used to dynamically modify the router configuration after mesh is configured.

Return

- `ESP_OK`
- `ESP_ERR_MESH_ARGUMENT`

Parameters

- [in] `router`: pointer to router configuration

esp_err_t **esp_mesh_get_router**(*mesh_router_t* **router*)

Get router configuration.

Return

- `ESP_OK`
- `ESP_ERR_MESH_ARGUMENT`

Parameters

- [out] `router`: pointer to router configuration

esp_err_t **esp_mesh_set_id**(const *mesh_addr_t* **id*)

Set mesh network ID.

Attention This API is used to dynamically modify the mesh network ID.

Return

- ESP_OK
- ESP_ERR_MESH_ARGUMENT: invalid argument

Parameters

- [in] id: pointer to mesh network ID

esp_err_t **esp_mesh_get_id** (*mesh_addr_t* *id)

Get mesh network ID.

Return

- ESP_OK
- ESP_ERR_MESH_ARGUMENT

Parameters

- [out] id: pointer to mesh network ID

esp_err_t **esp_mesh_set_type** (*mesh_type_t* type)

Designate device type over the mesh network.

- MESH_IDLE: designates a device as a self-organized node for a mesh network
- MESH_ROOT: designates the root node for a mesh network
- MESH_LEAF: designates a device as a standalone Wi-Fi station that connects to a parent
- MESH_STA: designates a device as a standalone Wi-Fi station that connects to a router

Return

- ESP_OK
- ESP_ERR_MESH_NOT_ALLOWED

Parameters

- [in] type: device type

mesh_type_t **esp_mesh_get_type** (void)

Get device type over mesh network.

Attention This API shall be called after having received the event MESH_EVENT_PARENT_CONNECTED.

Return mesh type

esp_err_t **esp_mesh_set_max_layer** (int max_layer)

Set network max layer value (max:25, default:25)

- Network max layer limits the max hop count.

Attention This API shall be called before mesh is started.

Return

- ESP_OK
- ESP_ERR_MESH_ARGUMENT
- ESP_ERR_MESH_NOT_ALLOWED

Parameters

- [in] max_layer: max layer value

int **esp_mesh_get_max_layer** (void)

Get max layer value.

Return max layer value

esp_err_t **esp_mesh_set_ap_password** (const uint8_t *pwd, int len)

Set mesh softAP password.

Attention This API shall be called before mesh is started.

Return

- ESP_OK
- ESP_ERR_MESH_ARGUMENT
- ESP_ERR_MESH_NOT_ALLOWED

Parameters

- [in] pwd: pointer to the password

- [in] len: password length

esp_err_t **esp_mesh_set_ap_authmode** (*wifi_auth_mode_t* authmode)

Set mesh softAP authentication mode.

Attention This API shall be called before mesh is started.

Return

- ESP_OK
- ESP_ERR_MESH_ARGUMENT
- ESP_ERR_MESH_NOT_ALLOWED

Parameters

- [in] authmode: authentication mode

wifi_auth_mode_t **esp_mesh_get_ap_authmode** (void)

Get mesh softAP authentication mode.

Return authentication mode

esp_err_t **esp_mesh_set_ap_connections** (int connections)

Set mesh softAP max connection value.

Attention This API shall be called before mesh is started.

Return

- ESP_OK
- ESP_ERR_MESH_ARGUMENT

Parameters

- [in] connections: the number of max connections

int **esp_mesh_get_ap_connections** (void)

Get mesh softAP max connection configuration.

Return the number of max connections

int **esp_mesh_get_layer** (void)

Get current layer value over the mesh network.

Attention This API shall be called after having received the event MESH_EVENT_PARENT_CONNECTED.

Return layer value

esp_err_t **esp_mesh_get_parent_bssid** (*mesh_addr_t* *bssid)

Get the parent BSSID.

Attention This API shall be called after having received the event MESH_EVENT_PARENT_CONNECTED.

Return

- ESP_OK
- ESP_FAIL

Parameters

- [out] bssid: pointer to parent BSSID

bool **esp_mesh_is_root** (void)

Return whether the device is the root node of the network.

Return true/false

esp_err_t **esp_mesh_set_self_organized** (bool enable, bool select_parent)

Enable/disable self-organized networking.

- Self-organized networking has three main functions: select the root node; find a preferred parent; initiate reconnection if a disconnection is detected.
- Self-organized networking is enabled by default.
- If self-organized is disabled, users should set a parent for the device via `esp_mesh_set_parent()`.

Attention This API is used to dynamically modify whether to enable the self organizing.

Return

- ESP_OK

- ESP_FAIL

Parameters

- [in] `enable`: enable or disable self-organized networking
- [in] `select_parent`: Only valid when self-organized networking is enabled.
 - if `select_parent` is set to true, the root will give up its mesh root status and search for a new parent like other non-root devices.

bool **esp_mesh_get_self_organized** (void)

Return whether enable self-organized networking or not.

Return true/false

esp_err_t **esp_mesh_waive_root** (const *mesh_vote_t* *`vote`, int `reason`)

Cause the root device to give up (waive) its mesh root status.

- A device is elected root primarily based on RSSI from the external router.
- If external router conditions change, users can call this API to perform a root switch.
- In this API, users could specify a desired root address to replace itself or specify an attempts value to ask current root to initiate a new round of voting. During the voting, a better root candidate would be expected to find to replace the current one.
- If no desired root candidate, the vote will try a specified number of attempts (at least 15). If no better root candidate is found, keep the current one. If a better candidate is found, the new better one will send a root switch request to the current root, current root will respond with a root switch acknowledgment.
- After that, the new candidate will connect to the router to be a new root, the previous root will disconnect with the router and choose another parent instead.

Root switch is completed with minimal disruption to the whole mesh network.

Attention This API is only called by the root.

Return

- ESP_OK
- ESP_ERR_MESH_QUEUE_FULL
- ESP_ERR_MESH_DISCARD
- ESP_FAIL

Parameters

- [in] `vote`: vote configuration
 - If this parameter is set NULL, the vote will perform the default 15 times.
 - Field percentage threshold is 0.9 by default.
 - Field `is_rc_specified` shall be false.
 - Field `attempts` shall be at least 15 times.
- [in] `reason`: only accept MESH_VOTE_REASON_ROOT_INITIATED for now

esp_err_t **esp_mesh_set_vote_percentage** (float `percentage`)

Set vote percentage threshold for approval of being a root.

- During the networking, only obtaining vote percentage reaches this threshold, the device could be a root.

Attention This API shall be called before mesh is started.

Return

- ESP_OK
- ESP_FAIL

Parameters

- [in] `percentage`: vote percentage threshold

float **esp_mesh_get_vote_percentage** (void)

Get vote percentage threshold for approval of being a root.

Return percentage threshold

esp_err_t **esp_mesh_set_ap_assoc_expire** (int `seconds`)

Set mesh softAP associate expired time (default:10 seconds)

- If mesh softAP hasn't received any data from an associated child within this time, mesh softAP will take this child inactive and disassociate it.
- If mesh softAP is encrypted, this value should be set a greater value, such as 30 seconds.

Return

- ESP_OK
- ESP_FAIL

Parameters

- [in] seconds: the expired time

int **esp_mesh_get_ap_assoc_expire** (void)

Get mesh softAP associate expired time.

Return seconds

int **esp_mesh_get_total_node_num** (void)

Get total number of devices in current network (including the root)

Attention The returned value might be incorrect when the network is changing.

Return total number of devices (including the root)

int **esp_mesh_get_routing_table_size** (void)

Get the number of devices in this device's sub-network (including self)

Return the number of devices over this device's sub-network (including self)

esp_err_t **esp_mesh_get_routing_table** (*mesh_addr_t* *mac, int len, int *size)

Get routing table of this device's sub-network (including itself)

Return

- ESP_OK
- ESP_ERR_MESH_ARGUMENT

Parameters

- [out] mac: pointer to routing table
- [in] len: routing table size(in bytes)
- [out] size: pointer to the number of devices in routing table (including itself)

esp_err_t **esp_mesh_post_toDS_state** (bool reachable)

Post the toDS state to the mesh stack.

Attention This API is only for the root.

Return

- ESP_OK
- ESP_FAIL

Parameters

- [in] reachable: this state represents whether the root is able to access external IP network

esp_err_t **esp_mesh_get_tx_pending** (*mesh_tx_pending_t* *pending)

Return the number of packets pending in the queue waiting to be sent by the mesh stack.

Return

- ESP_OK
- ESP_FAIL

Parameters

- [out] pending: pointer to the TX pending

esp_err_t **esp_mesh_get_rx_pending** (*mesh_rx_pending_t* *pending)

Return the number of packets available in the queue waiting to be received by applications.

Return

- ESP_OK
- ESP_FAIL

Parameters

- [out] pending: pointer to the RX pending

int **esp_mesh_available_txupQ_num** (const *mesh_addr_t* *addr, uint32_t *xseqno_in)

Return the number of packets could be accepted from the specified address.

Return the number of upQ for a certain address

Parameters

- [in] `addr`: self address or an associate children address
- [out] `xseqno_in`: sequence number of the last received packet from the specified address

esp_err_t **esp_mesh_set_xon_qsize** (int *qsize*)

Set the number of queue.

Attention This API shall be called before mesh is started.

Return

- ESP_OK
- ESP_FAIL

Parameters

- [in] `qsize`: default:32 (min:16)

int **esp_mesh_get_xon_qsize** (void)

Get queue size.

Return the number of queue

esp_err_t **esp_mesh_allow_root_conflicts** (bool *allowed*)

Set whether allow more than one root existing in one network.

Return

- ESP_OK
- ESP_WIFI_ERR_NOT_INIT
- ESP_WIFI_ERR_NOT_START

Parameters

- [in] `allowed`: allow or not

bool **esp_mesh_is_root_conflicts_allowed** (void)

Check whether allow more than one root to exist in one network.

Return true/false

esp_err_t **esp_mesh_set_group_id** (const *mesh_addr_t* **addr*, int *num*)

Set group ID addresses.

Return

- ESP_OK
- ESP_MESH_ERR_ARGUMENT

Parameters

- [in] `addr`: pointer to new group ID addresses
- [in] `num`: the number of group ID addresses

esp_err_t **esp_mesh_delete_group_id** (const *mesh_addr_t* **addr*, int *num*)

Delete group ID addresses.

Return

- ESP_OK
- ESP_MESH_ERR_ARGUMENT

Parameters

- [in] `addr`: pointer to deleted group ID address
- [in] `num`: the number of group ID addresses

int **esp_mesh_get_group_num** (void)

Get the number of group ID addresses.

Return the number of group ID addresses

esp_err_t **esp_mesh_get_group_list** (*mesh_addr_t* **addr*, int *num*)

Get group ID addresses.

Return

- ESP_OK
- ESP_MESH_ERR_ARGUMENT

Parameters

- [out] `addr`: pointer to group ID addresses

- [in] num: the number of group ID addresses

bool **esp_mesh_is_my_group** (const *mesh_addr_t* *addr)

Check whether the specified group address is my group.

Return true/false

esp_err_t **esp_mesh_set_capacity_num** (int num)

Set mesh network capacity (max:1000, default:300)

Attention This API shall be called before mesh is started.

Return

- ESP_OK
- ESP_ERR_MESH_NOT_ALLOWED
- ESP_MESH_ERR_ARGUMENT

Parameters

- [in] num: mesh network capacity

int **esp_mesh_get_capacity_num** (void)

Get mesh network capacity.

Return mesh network capacity

esp_err_t **esp_mesh_set_ie_crypto_funcs** (const *mesh_crypto_funcs_t* *crypto_funcs)

Set mesh IE crypto functions.

Attention This API can be called at any time after mesh is initialized.

Return

- ESP_OK

Parameters

- [in] crypto_funcs: crypto functions for mesh IE
 - If crypto_funcs is set to NULL, mesh IE is no longer encrypted.

esp_err_t **esp_mesh_set_ie_crypto_key** (const char *key, int len)

Set mesh IE crypto key.

Attention This API can be called at any time after mesh is initialized.

Return

- ESP_OK
- ESP_MESH_ERR_ARGUMENT

Parameters

- [in] key: ASCII crypto key
- [in] len: length in bytes, range:8-64

esp_err_t **esp_mesh_get_ie_crypto_key** (char *key, int len)

Get mesh IE crypto key.

Return

- ESP_OK
- ESP_MESH_ERR_ARGUMENT

Parameters

- [out] key: ASCII crypto key
- [in] len: length in bytes, range:8-64

esp_err_t **esp_mesh_set_root_healing_delay** (int delay_ms)

Set delay time before starting root healing.

Return

- ESP_OK

Parameters

- [in] delay_ms: delay time in milliseconds

int **esp_mesh_get_root_healing_delay** (void)

Get delay time before network starts root healing.

Return delay time in milliseconds

esp_err_t **esp_mesh_fix_root** (bool *enable*)

Enable network Fixed Root Setting.

- Enabling fixed root disables automatic election of the root node via voting.
- All devices in the network shall use the same Fixed Root Setting (enabled or disabled).
- If Fixed Root is enabled, users should make sure a root node is designated for the network.

Return

- ESP_OK

Parameters

- [in] *enable*: enable or not

bool **esp_mesh_is_root_fixed** (void)

Check whether network Fixed Root Setting is enabled.

- Enable/disable network Fixed Root Setting by API `esp_mesh_fix_root()`.
- Network Fixed Root Setting also changes with the “flag” value in parent networking IE.

Return true/false

esp_err_t **esp_mesh_set_parent** (const *wifi_config_t* **parent*, const *mesh_addr_t* **parent_mesh_id*, *mesh_type_t* *my_type*, int *my_layer*)

Set a specified parent for the device.

Attention This API can be called at any time after mesh is configured.

Return

- ESP_OK
- ESP_ERR_ARGUMENT
- ESP_ERR_MESH_NOT_CONFIG

Parameters

- [in] *parent*: parent configuration, the SSID and the channel of the parent are mandatory.
 - If the BSSID is set, make sure that the SSID and BSSID represent the same parent, otherwise the device will never find this specified parent.
- [in] *parent_mesh_id*: parent mesh ID,
 - If this value is not set, the original mesh ID is used.
- [in] *my_type*: mesh type
 - If the parent set for the device is the same as the router in the network configuration, then *my_type* shall set MESH_ROOT and *my_layer* shall set MESH_ROOT_LAYER.
- [in] *my_layer*: mesh layer
 - *my_layer* of the device may change after joining the network.
 - If *my_type* is set MESH_NODE, *my_layer* shall be greater than MESH_ROOT_LAYER.
 - If *my_type* is set MESH_LEAF, the device becomes a standalone Wi-Fi station and no longer has the ability to extend the network.

esp_err_t **esp_mesh_scan_get_ap_ie_len** (int **len*)

Get mesh networking IE length of one AP.

Return

- ESP_OK
- ESP_ERR_WIFI_NOT_INIT
- ESP_ERR_WIFI_ARG
- ESP_ERR_WIFI_FAIL

Parameters

- [out] *len*: mesh networking IE length

esp_err_t **esp_mesh_scan_get_ap_record** (*wifi_ap_record_t* **ap_record*, void **buffer*)

Get AP record.

Attention Different from `esp_wifi_scan_get_ap_records()`, this API only gets one of APs scanned each time. See “manual_networking” example.

Return

- ESP_OK
- ESP_ERR_WIFI_NOT_INIT
- ESP_ERR_WIFI_ARG

- ESP_ERR_WIFI_FAIL

Parameters

- [out] ap_record: pointer to one AP record
- [out] buffer: pointer to the mesh networking IE of this AP

esp_err_t **esp_mesh_flush_upstream_packets** (void)

Flush upstream packets pending in to_parent queue and to_parent_p2p queue.

Return

- ESP_OK

esp_err_t **esp_mesh_get_subnet_nodes_num** (const *mesh_addr_t* *child_mac, int *nodes_num)

Get the number of nodes in the subnet of a specific child.

Return

- ESP_OK
- ESP_ERR_MESH_NOT_START
- ESP_ERR_MESH_ARGUMENT

Parameters

- [in] child_mac: an associated child address of this device
- [out] nodes_num: pointer to the number of nodes in the subnet of a specific child

esp_err_t **esp_mesh_get_subnet_nodes_list** (const *mesh_addr_t* *child_mac, *mesh_addr_t* *nodes, int nodes_num)

Get nodes in the subnet of a specific child.

Return

- ESP_OK
- ESP_ERR_MESH_NOT_START
- ESP_ERR_MESH_ARGUMENT

Parameters

- [in] child_mac: an associated child address of this device
- [out] nodes: pointer to nodes in the subnet of a specific child
- [in] nodes_num: the number of nodes in the subnet of a specific child

esp_err_t **esp_mesh_disconnect** (void)

Disconnect from current parent.

Return

- ESP_OK

esp_err_t **esp_mesh_connect** (void)

Connect to current parent.

Return

- ESP_OK

esp_err_t **esp_mesh_flush_scan_result** (void)

Flush scan result.

Return

- ESP_OK

esp_err_t **esp_mesh_switch_channel** (const uint8_t *new_bssid, int csa_newchan, int csa_count)

Cause the root device to add Channel Switch Announcement Element (CSA IE) to beacon.

- Set the new channel
- Set how many beacons with CSA IE will be sent before changing a new channel
- Enable the channel switch function

Attention This API is only called by the root.

Return

- ESP_OK

Parameters

- [in] new_bssid: the new router BSSID if the router changes
- [in] csa_newchan: the new channel number to which the whole network is moving

- [in] `csa_count`: channel switch period (beacon count), unit is based on beacon interval of its softAP, the default value is 15.

`esp_err_t esp_mesh_get_router_bssid` (uint8_t **router_bssid*)

Get the router BSSID.

Return

- ESP_OK
- ESP_ERR_WIFI_NOT_INIT
- ESP_ERR_WIFI_ARG

Parameters

- [out] `router_bssid`: pointer to the router BSSID

int64_t `esp_mesh_get_tsf_time` (void)

Get the TSF time.

Return the TSF time

Unions

`union mesh_addr_t`

#include <esp_mesh.h> Mesh address.

Public Members

uint8_t `addr`[6]
mac address

mip_t `mip`
mip address

`union mesh_event_info_t`

#include <esp_mesh.h> Mesh event information.

Public Members

mesh_event_channel_switch_t `channel_switch`
channel switch

mesh_event_child_connected_t `child_connected`
child connected

mesh_event_child_disconnected_t `child_disconnected`
child disconnected

mesh_event_routing_table_change_t `routing_table`
routing table change

mesh_event_connected_t `connected`
parent connected

mesh_event_disconnected_t `disconnected`
parent disconnected

mesh_event_no_parent_found_t `no_parent`
no parent found

mesh_event_layer_change_t `layer_change`
layer change

mesh_event_toDS_state_t `toDS_state`

toDS state, devices shall check this state firstly before trying to send packets to external IP network. This state indicates right now whether the root is capable of sending packets out. If not, devices had better to wait until this state changes to be MESH_TODS_REACHABLE.

mesh_event_vote_started_t **vote_started**
vote started

mesh_event_root_address_t **root_addr**
root address

mesh_event_root_switch_req_t **switch_req**
root switch request

mesh_event_root_conflict_t **root_conflict**
other powerful root

mesh_event_root_fixed_t **root_fixed**
fixed root

mesh_event_scan_done_t **scan_done**
scan done

mesh_event_network_state_t **network_state**
network state, such as whether current mesh network has a root.

mesh_event_find_network_t **find_network**
network found that can join

mesh_event_router_switch_t **router_switch**
new router information

union mesh_rc_config_t
#include <esp_mesh.h> Vote address configuration.

Public Members

int **attempts**
max vote attempts before a new root is elected automatically by mesh network. (min:15, 15 by default)

mesh_addr_t **rc_addr**
a new root address specified by users for API `esp_mesh_waive_root()`

Structures

struct mip_t
IP address and port.

Public Members

ip4_addr_t **ip4**
IP address

uint16_t **port**
port

struct mesh_event_channel_switch_t
Channel switch information.

Public Members

uint8_t **channel**
new channel

struct mesh_event_connected_t
Parent connected information.

Public Members

wifi_event_sta_connected_t **connected**

parent information, same as Wi-Fi event SYSTEM_EVENT_STA_CONNECTED does

uint8_t **self_layer**

layer

struct mesh_event_no_parent_found_t

No parent found information.

Public Members

int **scan_times**

scan times being through

struct mesh_event_layer_change_t

Layer change information.

Public Members

uint8_t **new_layer**

new layer

struct mesh_event_vote_started_t

vote started information

Public Members

int **reason**

vote reason, vote could be initiated by children or by the root itself

int **attempts**

max vote attempts before stopped

mesh_addr_t **rc_addr**

root address specified by users via API esp_mesh_waive_root()

struct mesh_event_find_network_t

find a mesh network that this device can join

Public Members

uint8_t **channel**

channel number of the new found network

uint8_t **router_bssid**[6]

router BSSID

struct mesh_event_root_switch_req_t

Root switch request information.

Public Members

int **reason**

root switch reason, generally root switch is initialized by users via API esp_mesh_waive_root()

mesh_addr_t **rc_addr**

the address of root switch requester

struct mesh_event_root_conflict_t
Other powerful root address.

Public Members

int8_t rssi
rssi with router

uint16_t capacity
the number of devices in current network

uint8_t addr[6]
other powerful root address

struct mesh_event_routing_table_change_t
Routing table change.

Public Members

uint16_t rt_size_new
the new value

uint16_t rt_size_change
the changed value

struct mesh_event_root_fixed_t
Root fixed.

Public Members

bool is_fixed
status

struct mesh_event_scan_done_t
Scan done event information.

Public Members

uint8_t number
the number of APs scanned

struct mesh_event_network_state_t
Network state information.

Public Members

bool is_rootless
whether current mesh network has a root

struct mesh_opt_t
Mesh option.

Public Members

uint8_t type
option type

uint16_t len
option length

uint8_t *val
option value

struct mesh_data_t
Mesh data for esp_mesh_send() and esp_mesh_rcv()

Public Members

uint8_t *data
data

uint16_t size
data size

mesh_proto_t **proto**
data protocol

mesh_tos_t **tos**
data type of service

struct mesh_router_t
Router configuration.

Public Members

uint8_t ssid[32]
SSID

uint8_t ssid_len
length of SSID

uint8_t bssid[6]
BSSID, if this value is specified, users should also specify “allow_router_switch” .

uint8_t password[64]
password

bool allow_router_switch
if the BSSID is specified and this value is also set, when the router of this specified BSSID fails to be found after “fail” (mesh_attempts_t) times, the whole network is allowed to switch to another router with the same SSID. The new router might also be on a different channel. The default value is false. There is a risk that if the password is different between the new switched router and the previous one, the mesh network could be established but the root will never connect to the new switched router.

struct mesh_ap_cfg_t
Mesh softAP configuration.

Public Members

uint8_t password[64]
mesh softAP password

uint8_t max_connection
max number of stations allowed to connect in, max 10

struct mesh_cfg_t
Mesh initialization configuration.

Public Members

uint8_t **channel**

channel, the mesh network on

bool **allow_channel_switch**

if this value is set, when “fail” (mesh_attempts_t) times is reached, device will change to a full channel scan for a network that could join. The default value is false.

mesh_addr_t **mesh_id**

mesh network identification

mesh_router_t **router**

router configuration

mesh_ap_cfg_t **mesh_ap**

mesh softAP configuration

const mesh_crypto_funcs_t ***crypto_funcs**

crypto functions

struct **mesh_vote_t**

Vote.

Public Members

float **percentage**

vote percentage threshold for approval of being a root

bool **is_rc_specified**

if true, rc_addr shall be specified (Unimplemented). if false, attempts value shall be specified to make network start root election.

mesh_rc_config_t **config**

vote address configuration

struct **mesh_tx_pending_t**

The number of packets pending in the queue waiting to be sent by the mesh stack.

Public Members

int **to_parent**

to parent queue

int **to_parent_p2p**

to parent (P2P) queue

int **to_child**

to child queue

int **to_child_p2p**

to child (P2P) queue

int **mgmt**

management queue

int **broadcast**

broadcast and multicast queue

struct **mesh_rx_pending_t**

The number of packets available in the queue waiting to be received by applications.

Public Members

int **toDS**
to external DS

int **toSelf**
to self

Macros

MESH_ROOT_LAYER
root layer value

MESH_MTU
max transmit unit(in bytes)

MESH_MPS
max payload size(in bytes)

ESP_ERR_MESH_WIFI_NOT_START
Mesh error code definition.
Wi-Fi isn't started

ESP_ERR_MESH_NOT_INIT
mesh isn't initialized

ESP_ERR_MESH_NOT_CONFIG
mesh isn't configured

ESP_ERR_MESH_NOT_START
mesh isn't started

ESP_ERR_MESH_NOT_SUPPORT
not supported yet

ESP_ERR_MESH_NOT_ALLOWED
operation is not allowed

ESP_ERR_MESH_NO_MEMORY
out of memory

ESP_ERR_MESH_ARGUMENT
illegal argument

ESP_ERR_MESH_EXCEED_MTU
packet size exceeds MTU

ESP_ERR_MESH_TIMEOUT
timeout

ESP_ERR_MESH_DISCONNECTED
disconnected with parent on station interface

ESP_ERR_MESH_QUEUE_FAIL
queue fail

ESP_ERR_MESH_QUEUE_FULL
queue full

ESP_ERR_MESH_NO_PARENT_FOUND
no parent found to join the mesh network

ESP_ERR_MESH_NO_ROUTE_FOUND
no route found to forward the packet

ESP_ERR_MESH_OPTION_NULL
no option found

ESP_ERR_MESH_OPTION_UNKNOWN

unknown option

ESP_ERR_MESH_XON_NO_WINDOW

no window for software flow control on upstream

ESP_ERR_MESH_INTERFACE

low-level Wi-Fi interface error

ESP_ERR_MESH_DISCARD_DUPLICATE

discard the packet due to the duplicate sequence number

ESP_ERR_MESH_DISCARD

discard the packet

ESP_ERR_MESH_VOTING

vote in progress

ESP_ERR_MESH_XMIT

XMIT

ESP_ERR_MESH_QUEUE_READ

error in reading queue

ESP_ERR_MESH_RECV_RELEASE

release esp_mesh_rcv_toDS

MESH_DATA_ENC

Flags bitmap for esp_mesh_send() and esp_mesh_rcv()

data encrypted (Unimplemented)

MESH_DATA_P2P

point-to-point delivery over the mesh network

MESH_DATA_FROMDS

receive from external IP network

MESH_DATA_TODS

identify this packet is target to external IP network

MESH_DATA_NONBLOCK

esp_mesh_send() non-block

MESH_DATA_DROP

in the situation of the root having been changed, identify this packet can be dropped by new root

MESH_DATA_GROUP

identify this packet is target to a group address

MESH_OPT_SEND_GROUP

Option definitions for esp_mesh_send() and esp_mesh_rcv()

data transmission by group; used with esp_mesh_send() and shall have payload

MESH_OPT_RECV_DS_ADDR

return a remote IP address; used with esp_mesh_send() and esp_mesh_rcv()

MESH_ASSOC_FLAG_VOTE_IN_PROGRESS

Flag of mesh networking IE.

vote in progress

MESH_ASSOC_FLAG_NETWORK_FREE

no root in current network

MESH_ASSOC_FLAG_ROOTS_FOUND

root conflict is found

MESH_ASSOC_FLAG_ROOT_FIXED

fixed root

MESH_INIT_CONFIG_DEFAULT ()

Type Definitions

typedef *mesh_addr_t* **mesh_event_root_address_t**

Root address.

typedef *wifi_event_sta_disconnected_t* **mesh_event_disconnected_t**

Parent disconnected information.

typedef *wifi_event_ap_staconnected_t* **mesh_event_child_connected_t**

Child connected information.

typedef *wifi_event_ap_stadisconnected_t* **mesh_event_child_disconnected_t**

Child disconnected information.

typedef *wifi_event_sta_connected_t* **mesh_event_router_switch_t**

New router information.

Enumerations

enum **mesh_event_id_t**

Enumerated list of mesh event id.

Values:

MESH_EVENT_STARTED

mesh is started

MESH_EVENT_STOPPED

mesh is stopped

MESH_EVENT_CHANNEL_SWITCH

channel switch

MESH_EVENT_CHILD_CONNECTED

a child is connected on softAP interface

MESH_EVENT_CHILD_DISCONNECTED

a child is disconnected on softAP interface

MESH_EVENT_ROUTING_TABLE_ADD

routing table is changed by adding newly joined children

MESH_EVENT_ROUTING_TABLE_REMOVE

routing table is changed by removing leave children

MESH_EVENT_PARENT_CONNECTED

parent is connected on station interface

MESH_EVENT_PARENT_DISCONNECTED

parent is disconnected on station interface

MESH_EVENT_NO_PARENT_FOUND

no parent found

MESH_EVENT_LAYER_CHANGE

layer changes over the mesh network

MESH_EVENT_TODS_STATE

state represents whether the root is able to access external IP network

MESH_EVENT_VOTE_STARTED

the process of voting a new root is started either by children or by the root

MESH_EVENT_VOTE_STOPPED

the process of voting a new root is stopped

MESH_EVENT_ROOT_ADDRESS

the root address is obtained. It is posted by mesh stack automatically.

MESH_EVENT_ROOT_SWITCH_REQ

root switch request sent from a new voted root candidate

MESH_EVENT_ROOT_SWITCH_ACK

root switch acknowledgment responds the above request sent from current root

MESH_EVENT_ROOT_ASKED_YIELD

the root is asked yield by a more powerful existing root. If self organized is disabled and this device is specified to be a root by users, users should set a new parent for this device. if self organized is enabled, this device will find a new parent by itself, users could ignore this event.

MESH_EVENT_ROOT_FIXED

when devices join a network, if the setting of Fixed Root for one device is different from that of its parent, the device will update the setting the same as its parent's. Fixed Root Setting of each device is variable as that setting changes of the root.

MESH_EVENT_SCAN_DONE

if self-organized networking is disabled, user can call `esp_wifi_scan_start()` to trigger this event, and add the corresponding scan done handler in this event.

MESH_EVENT_NETWORK_STATE

network state, such as whether current mesh network has a root.

MESH_EVENT_STOP_RECONNECTION

the root stops reconnecting to the router and non-root devices stop reconnecting to their parents.

MESH_EVENT_FIND_NETWORK

when the channel field in mesh configuration is set to zero, mesh stack will perform a full channel scan to find a mesh network that can join, and return the channel value after finding it.

MESH_EVENT_ROUTER_SWITCH

if users specify BSSID of the router in mesh configuration, when the root connects to another router with the same SSID, this event will be posted and the new router information is attached.

MESH_EVENT_MAX

enum mesh_type_t

Device type.

Values:

MESH_IDLE

hasn't joined the mesh network yet

MESH_ROOT

the only sink of the mesh network. Has the ability to access external IP network

MESH_NODE

intermediate device. Has the ability to forward packets over the mesh network

MESH_LEAF

has no forwarding ability

MESH_STA

connect to router with a standalone Wi-Fi station mode, no network expansion capability

enum mesh_proto_t

Protocol of transmitted application data.

Values:

MESH_PROTO_BIN

binary

MESH_PROTO_HTTP

HTTP protocol

MESH_PROTO_JSON

JSON format

MESH_PROTO_MQTT

MQTT protocol

enum mesh_tos_t

For reliable transmission, mesh stack provides three type of services.

*Values:***MESH_TOS_P2P**

provide P2P (point-to-point) retransmission on mesh stack by default

MESH_TOS_E2E

provide E2E (end-to-end) retransmission on mesh stack (Unimplemented)

MESH_TOS_DEF

no retransmission on mesh stack

enum mesh_vote_reason_t

Vote reason.

*Values:***MESH_VOTE_REASON_ROOT_INITIATED = 1**

vote is initiated by the root

MESH_VOTE_REASON_CHILD_INITIATED

vote is initiated by children

enum mesh_disconnect_reason_t

Mesh disconnect reason code.

*Values:***MESH_REASON_CYCLIC = 100**

cyclic is detected

MESH_REASON_PARENT_IDLE

parent is idle

MESH_REASON_LEAF

the connected device is changed to a leaf

MESH_REASON_DIFF_ID

in different mesh ID

MESH_REASON_ROOTS

root conflict is detected

MESH_REASON_PARENT_STOPPED

parent has stopped the mesh

MESH_REASON_SCAN_FAIL

scan fail

MESH_REASON_IE_UNKNOWN

unknown IE

MESH_REASON_WAIVE_ROOT

waive root

MESH_REASON_PARENT_WORSE

parent with very poor RSSI

MESH_REASON_EMPTY_PASSWORD

use an empty password to connect to an encrypted parent

MESH_REASON_PARENT_UNENCRYPTED

connect to an unencrypted parent/router

enum mesh_event_toDS_state_t

The reachability of the root to a DS (distribute system)

Values:

MESH_TODS_UNREACHABLE

the root isn't able to access external IP network

MESH_TODS_REACHABLE

the root is able to access external IP network

Code examples for the Wi-Fi API are provided in the [wifi](#) directory of ESP-IDF examples.

Code examples for ESP-MESH are provided in the [mesh](#) directory of ESP-IDF examples.

2.2.2 Ethernet

Ethernet

Application Example

- Ethernet basic example: [ethernet/ethernet](#).
- Ethernet iperf example: [ethernet/iperf](#).

Ethernet Driver Model

- [esp_eth/include/esp_eth.h](#)

Ethernet Common Interface

- [esp_eth/include/esp_eth_com.h](#)

Ethernet MAC Interface

- [esp_eth/include/esp_eth_mac.h](#)

Ethernet PHY Interface

- [esp_eth/include/esp_eth_phy.h](#)

Ethernet PHY Common Registers

- [esp_eth/include/eth_phy_regs_struct.h](#)

API Reference - Driver Model

Header File

- [esp_eth/include/esp_eth.h](#)

Functions

esp_err_t **esp_eth_driver_install** (const *esp_eth_config_t* *config, *esp_eth_handle_t* *out_hdl)

Install Ethernet driver.

Return

- ESP_OK: install esp_eth driver successfully
- ESP_ERR_INVALID_ARG: install esp_eth driver failed because of some invalid argument
- ESP_ERR_NO_MEM: install esp_eth driver failed because there's no memory for driver
- ESP_FAIL: install esp_eth driver failed because some other error occurred

Parameters

- [in] config: configuration of the Ethernet driver
- [out] out_hdl: handle of Ethernet driver

esp_err_t **esp_eth_driver_uninstall** (*esp_eth_handle_t* hdl)

Uninstall Ethernet driver.

Note It's not recommended to uninstall Ethernet driver unless it won't get used any more in application code. To uninstall Ethernet driver, you have to make sure, all references to the driver are released. Ethernet driver can only be uninstalled successfully when reference counter equals to one.

Return

- ESP_OK: uninstall esp_eth driver successfully
- ESP_ERR_INVALID_ARG: uninstall esp_eth driver failed because of some invalid argument
- ESP_ERR_INVALID_STATE: uninstall esp_eth driver failed because it has more than one reference
- ESP_FAIL: uninstall esp_eth driver failed because some other error occurred

Parameters

- [in] hdl: handle of Ethernet driver

esp_err_t **esp_eth_start** (*esp_eth_handle_t* hdl)

Start Ethernet driver **ONLY** in standalone mode (i.e. without TCP/IP stack)

Note This API will start driver state machine and internal software timer (for checking link status).

Return

- ESP_OK: start esp_eth driver successfully
- ESP_ERR_INVALID_ARG: start esp_eth driver failed because of some invalid argument
- ESP_ERR_INVALID_STATE: start esp_eth driver failed because driver has started already
- ESP_FAIL: start esp_eth driver failed because some other error occurred

Parameters

- [in] hdl: handle of Ethernet driver

esp_err_t **esp_eth_stop** (*esp_eth_handle_t* hdl)

Stop Ethernet driver.

Note This function does the oppsite operation of esp_eth_start.

Return

- ESP_OK: stop esp_eth driver successfully
- ESP_ERR_INVALID_ARG: stop esp_eth driver failed because of some invalid argument
- ESP_ERR_INVALID_STATE: stop esp_eth driver failed because driver has not started yet
- ESP_FAIL: stop esp_eth driver failed because some other error occurred

Parameters

- [in] hdl: handle of Ethernet driver

esp_err_t **esp_eth_update_input_path** (*esp_eth_handle_t* hdl, *esp_err_t* (*stack_input)) *esp_eth_handle_t* hdl, uint8_t *buffer, uint32_t length, void *priv, void *priv) Update Ethernet data input path (i.e. specify where to pass the input buffer)

Note After install driver, Ethernet still don't know where to deliver the input buffer. In fact, this API registers a callback function which get invoked when Ethernet received new packets.

Return

- ESP_OK: update input path successfully
- ESP_ERR_INVALID_ARG: update input path failed because of some invalid argument
- ESP_FAIL: update input path failed because some other error occurred

Parameters

- [in] `hdl`: handle of Ethernet driver
- [in] `stack_input`: function pointer, which does the actual process on incoming packets
- [in] `priv`: private resource, which gets passed to `stack_input` callback without any modification

esp_err_t **esp_eth_transmit** (*esp_eth_handle_t* *hdl*, void **buf*, uint32_t *length*)

General Transmit.

Return

- `ESP_OK`: transmit frame buffer successfully
- `ESP_ERR_INVALID_ARG`: transmit frame buffer failed because of some invalid argument
- `ESP_FAIL`: transmit frame buffer failed because some other error occurred

Parameters

- [in] `hdl`: handle of Ethernet driver
- [in] `buf`: buffer of the packet to transfer
- [in] `length`: length of the buffer to transfer

esp_err_t **esp_eth_receive** (*esp_eth_handle_t* *hdl*, uint8_t **buf*, uint32_t **length*)

General Receive.

Note Before this function got invoked, the value of “length” should set by user, equals the size of buffer. After the function returned, the value of “length” means the real length of received data.

Return

- `ESP_OK`: receive frame buffer successfully
- `ESP_ERR_INVALID_ARG`: receive frame buffer failed because of some invalid argument
- `ESP_ERR_INVALID_SIZE`: input buffer size is not enough to hold the incoming data. in this case, value of returned “length” indicates the real size of incoming data.
- `ESP_FAIL`: receive frame buffer failed because some other error occurred

Parameters

- [in] `hdl`: handle of Ethernet driver
- [out] `buf`: buffer to preserve the received packet
- [out] `length`: length of the received packet

esp_err_t **esp_eth_ioctl** (*esp_eth_handle_t* *hdl*, *esp_eth_io_cmd_t* *cmd*, void **data*)

Misc IO function of Ethernet driver.

Return

- `ESP_OK`: process io command successfully
- `ESP_ERR_INVALID_ARG`: process io command failed because of some invalid argument
- `ESP_FAIL`: process io command failed because some other error occurred

Parameters

- [in] `hdl`: handle of Ethernet driver
- [in] `cmd`: IO control command
- [in] `data`: specified data for command

esp_err_t **esp_eth_increase_reference** (*esp_eth_handle_t* *hdl*)

Increase Ethernet driver reference.

Note Ethernet driver handle can be obtained by `os_timer`, `netif`, etc. It’s dangerous when thread A is using Ethernet but thread B uninstall the driver. Using reference counter can prevent such risk, but care should be taken, when you obtain Ethernet driver, this API must be invoked so that the driver won’t be uninstalled during your using time.

Return

- `ESP_OK`: increase reference successfully
- `ESP_ERR_INVALID_ARG`: increase reference failed because of some invalid argument

Parameters

- [in] `hdl`: handle of Ethernet driver

esp_err_t **esp_eth_decrease_reference** (*esp_eth_handle_t* *hdl*)

Decrease Ethernet driver reference.

Return

- ESP_OK: increase reference successfully
- ESP_ERR_INVALID_ARG: increase reference failed because of some invalid argument

Parameters

- [in] hdl: handle of Ethernet driver

Structures**struct esp_eth_config_t**

Configuration of Ethernet driver.

Public Members*esp_eth_mac_t* *mac

Ethernet MAC object.

esp_eth_phy_t *phy

Ethernet PHY object.

uint32_t **check_link_period_ms**

Period time of checking Ethernet link status.

esp_err_t (***stack_input**) (*esp_eth_handle_t* eth_handle, uint8_t *buffer, uint32_t length, void *priv)

Input frame buffer to user's stack.

Return

- ESP_OK: input frame buffer to upper stack successfully
- ESP_FAIL: error occurred when inputting buffer to upper stack

Parameters

- [in] eth_handle: handle of Ethernet driver
- [in] buffer: frame buffer that will get input to upper stack
- [in] length: length of the frame buffer

esp_err_t (***on_lowlevel_init_done**) (*esp_eth_handle_t* eth_handle)

Callback function invoked when lowlevel initialization is finished.

Return

- ESP_OK: process extra lowlevel initialization successfully
- ESP_FAIL: error occurred when processing extra lowlevel initialization

Parameters

- [in] eth_handle: handle of Ethernet driver

esp_err_t (***on_lowlevel_deinit_done**) (*esp_eth_handle_t* eth_handle)

Callback function invoked when lowlevel deinitialization is finished.

Return

- ESP_OK: process extra lowlevel deinitialization successfully
- ESP_FAIL: error occurred when processing extra lowlevel deinitialization

Parameters

- [in] eth_handle: handle of Ethernet driver

Macros**ETH_DEFAULT_CONFIG** (emac, ephy)

Default configuration for Ethernet driver.

Type Definitions**typedef** void ***esp_eth_handle_t**

Handle of Ethernet driver.

API Reference - Common Interface

Header File

- `esp_eth/include/esp_eth_com.h`

Functions

`esp_err_t esp_eth_detect_phy_addr(esp_eth_mediator_t *eth, uint32_t *detected_addr)`
Detect PHY address.

Return

- ESP_OK: detect phy address successfully
- ESP_ERR_INVALID_ARG: invalid parameter
- ESP_ERR_NOT_FOUND: can't detect any PHY device
- ESP_FAIL: detect phy address failed because some error occurred

Parameters

- [in] eth: mediator of Ethernet driver
- [out] detected_addr: a valid address after detection

Structures

`struct esp_eth_mediator_s`
Ethernet mediator.

Public Members

`esp_err_t (*phy_reg_read)(esp_eth_mediator_t *eth, uint32_t phy_addr, uint32_t phy_reg, uint32_t *reg_value)`

Read PHY register.

Return

- ESP_OK: read PHY register successfully
- ESP_FAIL: read PHY register failed because some error occurred

Parameters

- [in] eth: mediator of Ethernet driver
- [in] phy_addr: PHY Chip address (0~31)
- [in] phy_reg: PHY register index code
- [out] reg_value: PHY register value

`esp_err_t (*phy_reg_write)(esp_eth_mediator_t *eth, uint32_t phy_addr, uint32_t phy_reg, uint32_t reg_value)`

Write PHY register.

Return

- ESP_OK: write PHY register successfully
- ESP_FAIL: write PHY register failed because some error occurred

Parameters

- [in] eth: mediator of Ethernet driver
- [in] phy_addr: PHY Chip address (0~31)
- [in] phy_reg: PHY register index code
- [in] reg_value: PHY register value

`esp_err_t (*stack_input)(esp_eth_mediator_t *eth, uint8_t *buffer, uint32_t length)`
Deliver packet to upper stack.

Return

- ESP_OK: deliver packet to upper stack successfully
- ESP_FAIL: deliver packet failed because some error occurred

Parameters

- [in] eth: mediator of Ethernet driver
- [in] buffer: packet buffer
- [in] length: length of the packet

esp_err_t (***on_state_changed**) (*esp_eth_mediator_t* *eth, *esp_eth_state_t* state, void *args)
Callback on Ethernet state changed.

Return

- ESP_OK: process the new state successfully
- ESP_FAIL: process the new state failed because some error occurred

Parameters

- [in] eth: mediator of Ethernet driver
- [in] state: new state
- [in] args: optional argument for the new state

Macros**ETH_MAX_PAYLOAD_LEN**

Maximum Ethernet payload size.

ETH_MIN_PAYLOAD_LEN

Minimum Ethernet payload size.

ETH_HEADER_LEN

Ethernet frame header size: Dest addr(6 Bytes) + Src addr(6 Bytes) + length/type(2 Bytes)

ETH_CRC_LEN

Ethernet frame CRC length.

ETH_VLAN_TAG_LEN

Optional 802.1q VLAN Tag length.

ETH_JUMBO_FRAME_PAYLOAD_LEN

Jumbo frame payload size.

ETH_MAX_PACKET_SIZE

Maximum frame size (1522 Bytes)

ETH_MIN_PACKET_SIZE

Minimum frame size (64 Bytes)

Type Definitions

typedef struct *esp_eth_mediator_s* esp_eth_mediator_t
Ethernet mediator.

Enumerations

enum esp_eth_state_t
Ethernet driver state.

Values:

ETH_STATE_LLINIT
Lowlevel init done

ETH_STATE_DEINIT
Deinit done

ETH_STATE_LINK
Link status changed

ETH_STATE_SPEED
Speed updated

ETH_STATE_DUPLEX
Duplex updated

enum esp_eth_io_cmd_t
Command list for ioctl API.

Values:

ETH_CMD_G_MAC_ADDR
Get MAC address

ETH_CMD_S_MAC_ADDR
Set MAC address

ETH_CMD_G_PHY_ADDR
Get PHY address

ETH_CMD_S_PHY_ADDR
Set PHY address

ETH_CMD_G_SPEED
Get Speed

ETH_CMD_S_PROMISCUOUS
Set promiscuous mode

enum eth_link_t
Ethernet link status.

Values:

ETH_LINK_UP
Ethernet link is up

ETH_LINK_DOWN
Ethernet link is down

enum eth_speed_t
Ethernet speed.

Values:

ETH_SPEED_10M
Ethernet speed is 10Mbps

ETH_SPEED_100M
Ethernet speed is 100Mbps

enum eth_duplex_t
Ethernet duplex mode.

Values:

ETH_DUPLEX_HALF
Ethernet is in half duplex

ETH_DUPLEX_FULL
Ethernet is in full duplex

enum eth_event_t
Ethernet event declarations.

Values:

ETHERNET_EVENT_START
Ethernet driver start

ETHERNET_EVENT_STOP
Ethernet driver stop

ETHERNET_EVENT_CONNECTED
Ethernet got a valid link

ETHERNET_EVENT_DISCONNECTED
Ethernet lost a valid link

API Reference - MAC Interface

Header File

- `esp_eth/include/esp_eth_mac.h`

Structures

struct esp_eth_mac_s
Ethernet MAC.

Public Members

esp_err_t (***set_mediator**) (*esp_eth_mac_t* *mac, *esp_eth_mediator_t* *eth)
Set mediator for Ethernet MAC.

Return

- ESP_OK: set mediator for Ethernet MAC successfully
- ESP_ERR_INVALID_ARG: set mediator for Ethernet MAC failed because of invalid argument

Parameters

- [in] mac: Ethernet MAC instance
- [in] eth: Ethernet mediator

esp_err_t (***init**) (*esp_eth_mac_t* *mac)
Initialize Ethernet MAC.

Return

- ESP_OK: initialize Ethernet MAC successfully
- ESP_ERR_TIMEOUT: initialize Ethernet MAC failed because of timeout
- ESP_FAIL: initialize Ethernet MAC failed because some other error occurred

Parameters

- [in] mac: Ethernet MAC instance

esp_err_t (***deinit**) (*esp_eth_mac_t* *mac)
Deinitialize Ethernet MAC.

Return

- ESP_OK: deinitialize Ethernet MAC successfully
- ESP_FAIL: deinitialize Ethernet MAC failed because some error occurred

Parameters

- [in] mac: Ethernet MAC instance

esp_err_t (***start**) (*esp_eth_mac_t* *mac)
Start Ethernet MAC.

Return

- ESP_OK: start Ethernet MAC successfully
- ESP_FAIL: start Ethernet MAC failed because some other error occurred

Parameters

- [in] mac: Ethernet MAC instance

esp_err_t (***stop**) (*esp_eth_mac_t* *mac)
Stop Ethernet MAC.

Return

- ESP_OK: stop Ethernet MAC successfully
- ESP_FAIL: stop Ethernet MAC failed because some error occurred

Parameters

- [in] mac: Ethernet MAC instance

esp_err_t (***transmit**) (*esp_eth_mac_t* *mac, *uint8_t* *buf, *uint32_t* length)
Transmit packet from Ethernet MAC.

Return

- ESP_OK: transmit packet successfully

- ESP_ERR_INVALID_ARG: transmit packet failed because of invalid argument
- ESP_ERR_INVALID_STATE: transmit packet failed because of wrong state of MAC
- ESP_FAIL: transmit packet failed because some other error occurred

Parameters

- [in] mac: Ethernet MAC instance
- [in] buf: packet buffer to transmit
- [in] length: length of packet

esp_err_t (***receive**)(*esp_eth_mac_t* *mac, uint8_t *buf, uint32_t *length)

Receive packet from Ethernet MAC.

Note Memory of buf is allocated in the Layer2, make sure it get free after process.

Note Before this function got invoked, the value of “length” should set by user, equals the size of buffer.

After the function returned, the value of “length” means the real length of received data.

Return

- ESP_OK: receive packet successfully
- ESP_ERR_INVALID_ARG: receive packet failed because of invalid argument
- ESP_ERR_INVALID_SIZE: input buffer size is not enough to hold the incoming data. in this case, value of returned “length” indicates the real size of incoming data.
- ESP_FAIL: receive packet failed because some other error occurred

Parameters

- [in] mac: Ethernet MAC instance
- [out] buf: packet buffer which will preserve the received frame
- [out] length: length of the received packet

esp_err_t (***read_phy_reg**)(*esp_eth_mac_t* *mac, uint32_t phy_addr, uint32_t phy_reg, uint32_t *reg_value)

Read PHY register.

Return

- ESP_OK: read PHY register successfully
- ESP_ERR_INVALID_ARG: read PHY register failed because of invalid argument
- ESP_ERR_INVALID_STATE: read PHY register failed because of wrong state of MAC
- ESP_ERR_TIMEOUT: read PHY register failed because of timeout
- ESP_FAIL: read PHY register failed because some other error occurred

Parameters

- [in] mac: Ethernet MAC instance
- [in] phy_addr: PHY chip address (0~31)
- [in] phy_reg: PHY register index code
- [out] reg_value: PHY register value

esp_err_t (***write_phy_reg**)(*esp_eth_mac_t* *mac, uint32_t phy_addr, uint32_t phy_reg, uint32_t reg_value)

Write PHY register.

Return

- ESP_OK: write PHY register successfully
- ESP_ERR_INVALID_STATE: write PHY register failed because of wrong state of MAC
- ESP_ERR_TIMEOUT: write PHY register failed because of timeout
- ESP_FAIL: write PHY register failed because some other error occurred

Parameters

- [in] mac: Ethernet MAC instance
- [in] phy_addr: PHY chip address (0~31)
- [in] phy_reg: PHY register index code
- [in] reg_value: PHY register value

esp_err_t (***set_addr**)(*esp_eth_mac_t* *mac, uint8_t *addr)

Set MAC address.

Return

- ESP_OK: set MAC address successfully
- ESP_ERR_INVALID_ARG: set MAC address failed because of invalid argument

- ESP_FAIL: set MAC address failed because some other error occurred

Parameters

- [in] mac: Ethernet MAC instance
- [in] addr: MAC address

esp_err_t (***get_addr**) (*esp_eth_mac_t* *mac, uint8_t *addr)

Get MAC address.

Return

- ESP_OK: get MAC address successfully
- ESP_ERR_INVALID_ARG: get MAC address failed because of invalid argument
- ESP_FAIL: get MAC address failed because some other error occurred

Parameters

- [in] mac: Ethernet MAC instance
- [out] addr: MAC address

esp_err_t (***set_speed**) (*esp_eth_mac_t* *mac, *eth_speed_t* speed)

Set speed of MAC.

Return

- ESP_OK: set MAC speed successfully
- ESP_ERR_INVALID_ARG: set MAC speed failed because of invalid argument
- ESP_FAIL: set MAC speed failed because some other error occurred

Parameters

- [in] mac: Ethernet MAC instance
- [in] speed: MAC speed

esp_err_t (***set_duplex**) (*esp_eth_mac_t* *mac, *eth_duplex_t* duplex)

Set duplex mode of MAC.

Return

- ESP_OK: set MAC duplex mode successfully
- ESP_ERR_INVALID_ARG: set MAC duplex failed because of invalid argument
- ESP_FAIL: set MAC duplex failed because some other error occurred

Parameters

- [in] mac: Ethernet MAC instance
- [in] duplex: MAC duplex

esp_err_t (***set_link**) (*esp_eth_mac_t* *mac, *eth_link_t* link)

Set link status of MAC.

Return

- ESP_OK: set link status successfully
- ESP_ERR_INVALID_ARG: set link status failed because of invalid argument
- ESP_FAIL: set link status failed because some other error occurred

Parameters

- [in] mac: Ethernet MAC instance
- [in] link: Link status

esp_err_t (***set_promiscuous**) (*esp_eth_mac_t* *mac, bool enable)

Set promiscuous of MAC.

Return

- ESP_OK: set promiscuous mode successfully
- ESP_FAIL: set promiscuous mode failed because some error occurred

Parameters

- [in] mac: Ethernet MAC instance
- [in] enable: set true to enable promiscuous mode; set false to disable promiscuous mode

esp_err_t (***del**) (*esp_eth_mac_t* *mac)

Free memory of Ethernet MAC.

Return

- ESP_OK: free Ethernet MAC instance successfully

- `ESP_FAIL`: free Ethernet MAC instance failed because some error occurred

Parameters

- `[in] mac`: Ethernet MAC instance

struct eth_mac_config_t

Configuration of Ethernet MAC object.

Public Members

`uint32_t sw_reset_timeout_ms`

Software reset timeout value (Unit: ms)

`uint32_t rx_task_stack_size`

Stack size of the receive task

`uint32_t rx_task_prio`

Priority of the receive task

`int smi_mdc_gpio_num`

SMI MDC GPIO number

`int smi_mdio_gpio_num`

SMI MDIO GPIO number

`uint32_t flags`

Flags that specify extra capability for mac driver

Macros

ETH_MAC_FLAG_WORK_WITH_CACHE_DISABLE

MAC driver can work when cache is disabled

ETH_MAC_FLAG_PIN_TO_CORE

Pin MAC task to the CPU core where driver installation happened

ETH_MAC_DEFAULT_CONFIG()

Default configuration for Ethernet MAC object.

Type Definitions

typedef struct esp_eth_mac_s esp_eth_mac_t

Ethernet MAC.

API Reference - PHY Interface

Header File

- [esp_eth/include/esp_eth_phy.h](#)

Functions

`esp_eth_phy_t *esp_eth_phy_new_ip101(const eth_phy_config_t *config)`

Create a PHY instance of IP101.

Return

- instance: create PHY instance successfully
- NULL: create PHY instance failed because some error occurred

Parameters

- `[in] config`: configuration of PHY

`esp_eth_phy_t *esp_eth_phy_new_rt18201(const eth_phy_config_t *config)`

Create a PHY instance of RTL8201.

Return

- instance: create PHY instance successfully
- NULL: create PHY instance failed because some error occurred

Parameters

- [in] config: configuration of PHY

esp_eth_phy_t ***esp_eth_phy_new_lan8720** (const *eth_phy_config_t* *config)

Create a PHY instance of LAN8720.

Return

- instance: create PHY instance successfully
- NULL: create PHY instance failed because some error occurred

Parameters

- [in] config: configuration of PHY

esp_eth_phy_t ***esp_eth_phy_new_dp83848** (const *eth_phy_config_t* *config)

Create a PHY instance of DP83848.

Return

- instance: create PHY instance successfully
- NULL: create PHY instance failed because some error occurred

Parameters

- [in] config: configuration of PHY

Structures

struct esp_eth_phy_s

Ethernet PHY.

Public Members

esp_err_t (***set_mediator**) (*esp_eth_phy_t* *phy, *esp_eth_mediator_t* *mediator)

Set mediator for PHY.

Return

- ESP_OK: set mediator for Ethernet PHY instance successfully
- ESP_ERR_INVALID_ARG: set mediator for Ethernet PHY instance failed because of some invalid arguments

Parameters

- [in] phy: Ethernet PHY instance
- [in] mediator: mediator of Ethernet driver

esp_err_t (***reset**) (*esp_eth_phy_t* *phy)

Software Reset Ethernet PHY.

Return

- ESP_OK: reset Ethernet PHY successfully
- ESP_FAIL: reset Ethernet PHY failed because some error occurred

Parameters

- [in] phy: Ethernet PHY instance

esp_err_t (***reset_hw**) (*esp_eth_phy_t* *phy)

Hardware Reset Ethernet PHY.

Note Hardware reset is mostly done by pull down and up PHY' s nRST pin

Return

- ESP_OK: reset Ethernet PHY successfully
- ESP_FAIL: reset Ethernet PHY failed because some error occurred

Parameters

- [in] phy: Ethernet PHY instance

esp_err_t (***init**) (*esp_eth_phy_t* *phy)

Initialize Ethernet PHY.

Return

- ESP_OK: initialize Ethernet PHY successfully
- ESP_FAIL: initialize Ethernet PHY failed because some error occurred

Parameters

- [in] phy: Ethernet PHY instance

esp_err_t (***deinit**)(*esp_eth_phy_t* *phy)

Deinitialize Ethernet PHY.

Return

- ESP_OK: deinitialize Ethernet PHY successfully
- ESP_FAIL: deinitialize Ethernet PHY failed because some error occurred

Parameters

- [in] phyL: Ethernet PHY instance

esp_err_t (***negotiate**)(*esp_eth_phy_t* *phy)

Start auto negotiation.

Return

- ESP_OK: restart auto negotiation successfully
- ESP_FAIL: restart auto negotiation failed because some error occurred

Parameters

- [in] phy: Ethernet PHY instance

esp_err_t (***get_link**)(*esp_eth_phy_t* *phy)

Get Ethernet PHY link status.

Return

- ESP_OK: get Ethernet PHY link status successfully
- ESP_FAIL: get Ethernet PHY link status failed because some error occurred

Parameters

- [in] phy: Ethernet PHY instance

esp_err_t (***pwrcctl**)(*esp_eth_phy_t* *phy, bool enable)

Power control of Ethernet PHY.

Return

- ESP_OK: control Ethernet PHY power successfully
- ESP_FAIL: control Ethernet PHY power failed because some error occurred

Parameters

- [in] phy: Ethernet PHY instance
- [in] enable: set true to power on Ethernet PHY; set false to power off Ethernet PHY

esp_err_t (***set_addr**)(*esp_eth_phy_t* *phy, uint32_t addr)

Set PHY chip address.

Return

- ESP_OK: set Ethernet PHY address successfully
- ESP_FAIL: set Ethernet PHY address failed because some error occurred

Parameters

- [in] phy: Ethernet PHY instance
- [in] addr: PHY chip address

esp_err_t (***get_addr**)(*esp_eth_phy_t* *phy, uint32_t *addr)

Get PHY chip address.

Return

- ESP_OK: get Ethernet PHY address successfully
- ESP_ERR_INVALID_ARG: get Ethernet PHY address failed because of invalid argument

Parameters

- [in] phy: Ethernet PHY instance
- [out] addr: PHY chip address

esp_err_t (***del**)(*esp_eth_phy_t* *phy)

Free memory of Ethernet PHY instance.

Return

- ESP_OK: free PHY instance successfully
- ESP_FAIL: free PHY instance failed because some error occurred

Parameters

- [in] phy: Ethernet PHY instance

struct eth_phy_config_t
Ethernet PHY configuration.

Public Members

int32_t phy_addr

PHY address, set -1 to enable PHY address detection at initialization stage

uint32_t reset_timeout_ms

Reset timeout value (Unit: ms)

uint32_t autonego_timeout_ms

Auto-negotiation timeout value (Unit: ms)

int reset_gpio_num

Reset GPIO number, -1 means no hardware reset

Macros

ESP_ETH_PHY_ADDR_AUTO

ETH_PHY_DEFAULT_CONFIG ()

Default configuration for Ethernet PHY object.

Type Definitions

typedef struct esp_eth_phy_s esp_eth_phy_t
Ethernet PHY.

API Reference - Glue for esp_netif**Header File**

- [esp_eth/include/esp_eth_netif_glue.h](#)

Functions

void *esp_eth_new_netif_glue (esp_eth_handle_t eth_hdl)
Create a netif glue for Ethernet driver.

Note netif glue is used to attach io driver to TCP/IP netif

Return glue object, which inherits esp_netif_driver_base_t

Parameters

- eth_hdl: Ethernet driver handle

esp_err_t esp_eth_del_netif_glue (void *glue)
Delete netif glue of Ethernet driver.

Return -ESP_OK: delete netif glue successfully

Parameters

- glue: netif glue

esp_err_t esp_eth_set_default_handlers (void *esp_netif)
Register default IP layer handlers for Ethernet.

Note : Ethernet handle might not yet properly initialized when setting up these default handlers

Return

- ESP_ERR_INVALID_ARG: invalid parameter (esp_netif is NULL)

- ESP_OK: set default IP layer handlers successfully
- others: other failure occurred during register esp_event handler

Parameters

- [in] esp_netif: esp network interface handle created for Ethernet driver

esp_err_t **esp_eth_clear_default_handlers** (void *esp_netif)

Unregister default IP layer handlers for Ethernet.

Return

- ESP_ERR_INVALID_ARG: invalid parameter (esp_netif is NULL)
- ESP_OK: clear default IP layer handlers successfully
- others: other failure occurred during unregister esp_event handler

Parameters

- [in] esp_netif: esp network interface handle created for Ethernet driver

Code examples for the Ethernet API are provided in the [ethernet](#) directory of ESP-IDF examples.

2.2.3 IP Network Layer

ESP-NETIF

The purpose of ESP-NETIF library is twofold:

- It provides an abstraction layer for the application on top of the TCP/IP stack. This will allow applications to choose between IP stacks in the future.
- The APIs it provides are thread safe, even if the underlying TCP/IP stack APIs are not.

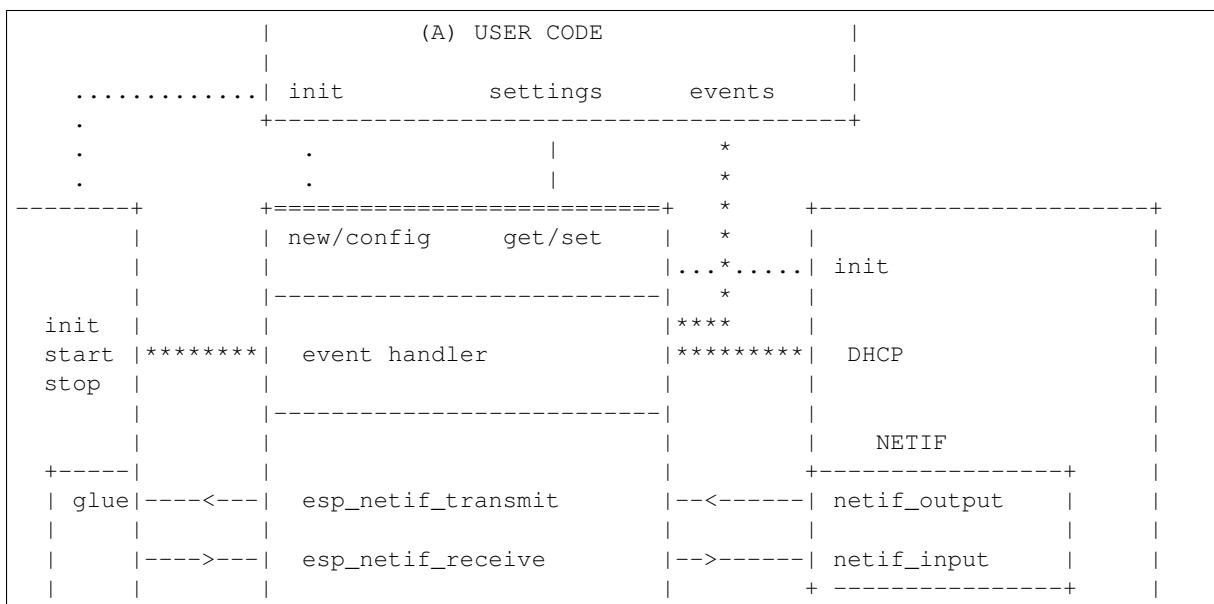
ESP-IDF currently implements ESP-NETIF for the lwIP TCP/IP stack only. However, the adapter itself is TCP/IP implementation agnostic and different implementations are possible.

Some ESP-NETIF API functions are intended to be called by application code, for example to get/set interface IP addresses, configure DHCP. Other functions are intended for internal ESP-IDF use by the network driver layer.

In many cases, applications do not need to call ESP-NETIF APIs directly as they are called from the default network event handlers.

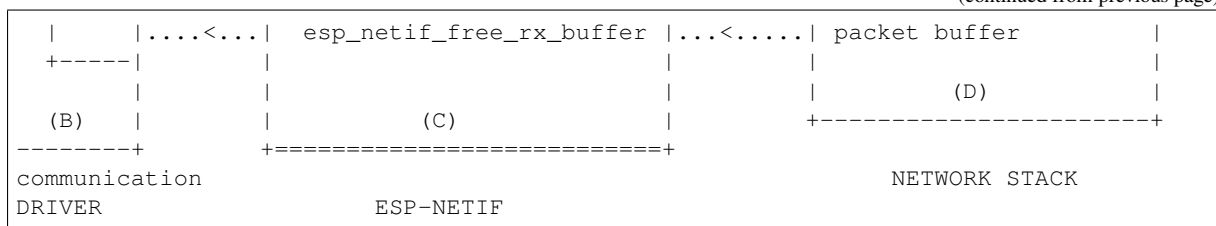
ESP-NETIF component is a successor of the tcpip_adapter, former network interface abstraction, which has become deprecated since IDF v4.1. Please refer to the [TCP/IP Adapter Migration Guide](#) section in case existing applications to be ported to use the esp-netif API instead.

ESP-NETIF architecture



(continues on next page)

(continued from previous page)



Data and event flow in the diagram

- Initialization line from user code to ESP-NETIF and communication driver
- ---<--->--- Data packets going from communication media to TCP/IP stack and back
- ***** Events aggregated in ESP-NETIF propagates to driver, user code and network stack
- | User settings and runtime configuration

ESP-NETIF interaction

A) User code, boiler plate Overall application interaction with a specific IO driver for communication media and configured TCP/IP network stack is abstracted using ESP-NETIF APIs and outlined as below:

A) Initialization code

- 1) Initializes IO driver
- 2) Creates a new instance of ESP-NETIF and configure with
 - ESP-NETIF specific options (flags, behaviour, name)
 - Network stack options (netif init and input functions, not publicly available)
 - IO driver specific options (transmit, free rx buffer functions, IO driver handle)
- 3) Attaches the IO driver handle to the ESP-NETIF instance created in the above steps
- 4) Configures event handlers
 - use default handlers for common interfaces defined in IO drivers; or define a specific handlers for customised behaviour/new interfaces
 - register handlers for app related events (such as IP lost/acquired)

B) Interaction with network interfaces using ESP-NETIF API

- Getting and setting TCP/IP related parameters (DHCP, IP, etc)
- Receiving IP events (connect/disconnect)
- Controlling application lifecycle (set interface up/down)

B) Communication driver, IO driver, media driver Communication driver plays these two important roles in relation with ESP-NETIF:

- 1) Event handlers: Define behaviour patterns of interaction with ESP-NETIF (for example: ethernet link-up -> turn netif on)
- 2) Glue IO layer: Adapts the input/output functions to use ESP-NETIF transmit, receive and free receive buffer
 - Installs driver_transmit to appropriate ESP-NETIF object, so that outgoing packets from network stack are passed to the IO driver
 - Calls `esp_netif_receive()` to pass incoming data to network stack

C) ESP-NETIF, former tcpip_adapter ESP-NETIF is an intermediary between an IO driver and a network stack, connecting packet data path between these two. As that it provides a set of interfaces for attaching a driver to ESP-NETIF object (runtime) and configuring a network stack (compile time). In addition to that a set of API is provided to control network interface lifecycle and its TCP/IP properties. As an overview, the ESP-NETIF public interface could be divided into these 6 groups:

- 1) Initialization APIs (to create and configure ESP-NETIF instance)
- 2) Input/Output API (for passing data between IO driver and network stack)
- 3) Event or Action API
 - Used for network interface lifecycle management
 - ESP-NETIF provides building blocks for designing event handlers
- 4) Setters and Getters for basic network interface properties
- 5) Network stack abstraction: enabling user interaction with TCP/IP stack
 - Set interface up or down
 - DHCP server and client API
 - DNS API
- 6) Driver conversion utilities

D) Network stack Network stack has no public interaction with application code with regard to public interfaces and shall be fully abstracted by ESP-NETIF API.

ESP-NETIF programmer' s manual Please refer to the example section for basic initialization of default interfaces:

- WiFi Station: [wifi/getting_started/station/main/station_example_main.c](#)
- WiFi Access Point: [wifi/getting_started/softAP/main/softap_example_main.c](#)
- Ethernet: [ethernet/basic/main/ethernet_example_main.c](#)

For more specific cases please consult this guide: [ESP-NETIF Custom I/O Driver](#).

WiFi default initialization The initialization code as well as registering event handlers for default interfaces, such as softAP and station, are provided in two separate APIs to facilitate simple startup code for most applications:

- [esp_netif_create_default_wifi_ap\(\)](#)
- [esp_netif_create_default_wifi_sta\(\)](#)

Please note that these functions return the `esp_netif` handle, i.e. a pointer to a network interface object allocated and configured with default settings, which as a consequence, means that:

- The created object has to be destroyed if a network de-initialization is provided by an application.
- These *default* interfaces must not be created multiple times, unless the created handle is deleted using [esp_netif_destroy\(\)](#).
- When using Wifi in AP+STA mode, both these interfaces has to be created.

API Reference

Header File

- [esp_netif/include/esp_netif.h](#)

Functions

`esp_err_t esp_netif_init` (void)

Initialize the underlying TCP/IP stack.

Return

- ESP_OK on success
- ESP_FAIL if initializing failed

Note This function should be called exactly once from application code, when the application starts up.

`esp_err_t esp_netif_deinit` (void)

Deinitialize the esp-netif component (and the underlying TCP/IP stack)

Note: Deinitialization is not supported yet

Return

- ESP_ERR_INVALID_STATE if esp_netif not initialized
- ESP_ERR_NOT_SUPPORTED otherwise

esp_netif_t ***esp_netif_new** (const esp_netif_config_t *esp_netif_config)
Creates an instance of new esp-netif object based on provided config.

Return

- pointer to esp-netif object on success
- NULL otherwise

Parameters

- [in] esp_netif_config: pointer esp-netif configuration

void **esp_netif_destroy** (esp_netif_t *esp_netif)
Destroys the esp_netif object.

Parameters

- [in] esp_netif: pointer to the object to be deleted

esp_err_t **esp_netif_set_driver_config** (esp_netif_t *esp_netif, const esp_netif_driver_ifconfig_t *driver_config)
Configures driver related options of esp_netif object.

Return

- ESP_OK on success
- ESP_ERR_ESP_NETIF_INVALID_PARAMS if invalid parameters provided

Parameters

- [inout] esp_netif: pointer to the object to be configured
- [in] driver_config: pointer esp-netif io driver related configuration

esp_err_t **esp_netif_attach** (esp_netif_t *esp_netif, esp_netif_io_driver_handle driver_handle)
Attaches esp_netif instance to the io driver handle.

Calling this function enables connecting specific esp_netif object with already initialized io driver to update esp_netif object with driver specific configuration (i.e. calls post_attach callback, which typically sets io driver callbacks to esp_netif instance and starts the driver)

Return

- ESP_OK on success
- ESP_ERR_ESP_NETIF_DRIVER_ATTACH_FAILED if driver's post_attach callback failed

Parameters

- [inout] esp_netif: pointer to esp_netif object to be attached
- [in] driver_handle: pointer to the driver handle

esp_err_t **esp_netif_receive** (esp_netif_t *esp_netif, void *buffer, size_t len, void *eb)
Passes the raw packets from communication media to the appropriate TCP/IP stack.

This function is called from the configured (peripheral) driver layer. The data are then forwarded as frames to the TCP/IP stack.

Return

- ESP_OK

Parameters

- [in] esp_netif: Handle to esp-netif instance
- [in] buffer: Received data
- [in] len: Length of the data frame
- [in] eb: Pointer to internal buffer (used in Wi-Fi driver)

void **esp_netif_action_start** (void *esp_netif, esp_event_base_t base, int32_t event_id, void *data)
Default building block for network interface action upon IO driver start event Creates network interface, if AUTOUP enabled turns the interface on, if DHCP enabled starts dhcp server.

Note This API can be directly used as event handler

Parameters

- [in] esp_netif: Handle to esp-netif instance

- base:
- event_id:
- data:

void **esp_netif_action_stop** (void **esp_netif*, *esp_event_base_t* base, int32_t *event_id*, void **data*)
Default building block for network interface action upon IO driver stop event.

Note This API can be directly used as event handler

Parameters

- [in] *esp_netif*: Handle to esp-netif instance
- base:
- event_id:
- data:

void **esp_netif_action_connected** (void **esp_netif*, *esp_event_base_t* base, int32_t *event_id*, void **data*)
Default building block for network interface action upon IO driver connected event.

Note This API can be directly used as event handler

Parameters

- [in] *esp_netif*: Handle to esp-netif instance
- base:
- event_id:
- data:

void **esp_netif_action_disconnected** (void **esp_netif*, *esp_event_base_t* base, int32_t *event_id*, void **data*)
Default building block for network interface action upon IO driver disconnected event.

Note This API can be directly used as event handler

Parameters

- [in] *esp_netif*: Handle to esp-netif instance
- base:
- event_id:
- data:

void **esp_netif_action_got_ip** (void **esp_netif*, *esp_event_base_t* base, int32_t *event_id*, void **data*)
Default building block for network interface action upon network got IP event.

Note This API can be directly used as event handler

Parameters

- [in] *esp_netif*: Handle to esp-netif instance
- base:
- event_id:
- data:

esp_err_t **esp_netif_set_mac** (*esp_netif_t* **esp_netif*, uint8_t *mac*[])

Set the mac address for the interface instance.

Return ESP_OK

Parameters

- [in] *esp_netif*: Handle to esp-netif instance
- [in] *mac*: Desired mac address for the related network interface

esp_err_t **esp_netif_set_hostname** (*esp_netif_t* **esp_netif*, const char **hostname*)

Set the hostname of an interface.

The configured hostname overrides the default configuration value CONFIG_LWIP_LOCAL_HOSTNAME. Please note that when the hostname is altered after interface started/connected the changes would only be reflected once the interface restarts/reconnects

Return

- ESP_OK - success
- ESP_ERR_ESP_NETIF_IF_NOT_READY - interface status error

- ESP_ERR_ESP_NETIF_INVALID_PARAMS - parameter error

Parameters

- [in] `esp_netif`: Handle to esp-netif instance
- [in] `hostname`: New hostname for the interface. Maximum length 32 bytes.

esp_err_t **esp_netif_get_hostname** (`esp_netif_t *esp_netif`, **const** char ***hostname*)

Get interface hostname.

Return

- ESP_OK - success
- ESP_ERR_ESP_NETIF_IF_NOT_READY - interface status error
- ESP_ERR_ESP_NETIF_INVALID_PARAMS - parameter error

Parameters

- [in] `esp_netif`: Handle to esp-netif instance
- [out] `hostname`: Returns a pointer to the hostname. May be NULL if no hostname is set. If set non-NULL, pointer remains valid (and string may change if the hostname changes).

bool **esp_netif_is_netif_up** (`esp_netif_t *esp_netif`)

Test if supplied interface is up or down.

Return

- true - Interface is up
- false - Interface is down

Parameters

- [in] `esp_netif`: Handle to esp-netif instance

esp_err_t **esp_netif_get_ip_info** (`esp_netif_t *esp_netif`, `esp_netif_ip_info_t *ip_info`)

Get interface's IP address information.

If the interface is up, IP information is read directly from the TCP/IP stack. If the interface is down, IP information is read from a copy kept in the ESP-NETIF instance

Return

- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS

Parameters

- [in] `esp_netif`: Handle to esp-netif instance
- [out] `ip_info`: If successful, IP information will be returned in this argument.

esp_err_t **esp_netif_get_old_ip_info** (`esp_netif_t *esp_netif`, `esp_netif_ip_info_t *ip_info`)

Get interface's old IP information.

Returns an "old" IP address previously stored for the interface when the valid IP changed.

If the IP lost timer has expired (meaning the interface was down for longer than the configured interval) then the old IP information will be zero.

Return

- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS

Parameters

- [in] `esp_netif`: Handle to esp-netif instance
- [out] `ip_info`: If successful, IP information will be returned in this argument.

esp_err_t **esp_netif_set_ip_info** (`esp_netif_t *esp_netif`, **const** `esp_netif_ip_info_t *ip_info`)

Set interface's IP address information.

This function is mainly used to set a static IP on an interface.

If the interface is up, the new IP information is set directly in the TCP/IP stack.

The copy of IP information kept in the ESP-NETIF instance is also updated (this copy is returned if the IP is queried while the interface is still down.)

Note DHCP client/server must be stopped (if enabled for this interface) before setting new IP information.

Note Calling this interface for may generate a SYSTEM_EVENT_STA_GOT_IP or SYSTEM_EVENT_ETH_GOT_IP event.

Return

- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
- ESP_ERR_ESP_NETIF_DHCP_NOT_STOPPED If DHCP server or client is still running

Parameters

- [in] `esp_netif`: Handle to esp-netif instance
- [in] `ip_info`: IP information to set on the specified interface

esp_err_t **esp_netif_set_old_ip_info**(`esp_netif_t *esp_netif`, **const** `esp_netif_ip_info_t *ip_info`)

Set interface old IP information.

This function is called from the DHCP client (if enabled), before a new IP is set. It is also called from the default handlers for the SYSTEM_EVENT_STA_CONNECTED and SYSTEM_EVENT_ETH_CONNECTED events.

Calling this function stores the previously configured IP, which can be used to determine if the IP changes in the future.

If the interface is disconnected or down for too long, the “IP lost timer” will expire (after the configured interval) and set the old IP information to zero.

Return

- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS

Parameters

- [in] `esp_netif`: Handle to esp-netif instance
- [in] `ip_info`: Store the old IP information for the specified interface

int **esp_netif_get_netif_impl_index**(`esp_netif_t *esp_netif`)

Get net interface index from network stack implementation.

Note This index could be used in `setsockopt()` to bind socket with multicast interface

Return implementation specific index of interface represented with supplied `esp_netif`

Parameters

- [in] `esp_netif`: Handle to esp-netif instance

esp_err_t **esp_netif_dhcps_option**(`esp_netif_t *esp_netif`, `esp_netif_dhcp_option_mode_t opt_op`, `esp_netif_dhcp_option_id_t opt_id`, `void *opt_val`, `uint32_t opt_len`)

Set or Get DHCP server option.

Return

- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
- ESP_ERR_ESP_NETIF_DHCP_ALREADY_STOPPED
- ESP_ERR_ESP_NETIF_DHCP_ALREADY_STARTED

Parameters

- [in] `esp_netif`: Handle to esp-netif instance
- [in] `opt_op`: ESP_NETIF_OP_SET to set an option, ESP_NETIF_OP_GET to get an option.
- [in] `opt_id`: Option index to get or set, must be one of the supported enum values.
- [inout] `opt_val`: Pointer to the option parameter.
- [in] `opt_len`: Length of the option parameter.

esp_err_t **esp_netif_dhcpc_option**(`esp_netif_t *esp_netif`, `esp_netif_dhcp_option_mode_t opt_op`, `esp_netif_dhcp_option_id_t opt_id`, `void *opt_val`, `uint32_t opt_len`)

Set or Get DHCP client option.

Return

- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS

- ESP_ERR_ESP_NETIF_DHCP_ALREADY_STOPPED
- ESP_ERR_ESP_NETIF_DHCP_ALREADY_STARTED

Parameters

- [in] `esp_netif`: Handle to esp-netif instance
- [in] `opt_op`: ESP_NETIF_OP_SET to set an option, ESP_NETIF_OP_GET to get an option.
- [in] `opt_id`: Option index to get or set, must be one of the supported enum values.
- [inout] `opt_val`: Pointer to the option parameter.
- [in] `opt_len`: Length of the option parameter.

esp_err_t **esp_netif_dhcpc_start** (*esp_netif_t *esp_netif*)

Start DHCP client (only if enabled in interface object)

Note The default event handlers for the SYSTEM_EVENT_STA_CONNECTED and SYSTEM_EVENT_ETH_CONNECTED events call this function.

Return

- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
- ESP_ERR_ESP_NETIF_DHCP_ALREADY_STARTED
- ESP_ERR_ESP_NETIF_DHCPC_START_FAILED

Parameters

- [in] `esp_netif`: Handle to esp-netif instance

esp_err_t **esp_netif_dhcpc_stop** (*esp_netif_t *esp_netif*)

Stop DHCP client (only if enabled in interface object)

Note Calling `action_netif_stop()` will also stop the DHCP Client if it is running.

Return

- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
- ESP_ERR_ESP_NETIF_DHCP_ALREADY_STOPPED
- ESP_ERR_ESP_NETIF_IF_NOT_READY

Parameters

- [in] `esp_netif`: Handle to esp-netif instance

esp_err_t **esp_netif_dhcpc_get_status** (*esp_netif_t *esp_netif*, *esp_netif_dhcp_status_t *status*)

Get DHCP client status.

Return

- ESP_OK

Parameters

- [in] `esp_netif`: Handle to esp-netif instance
- [out] `status`: If successful, the status of DHCP client will be returned in this argument.

esp_err_t **esp_netif_dhcps_get_status** (*esp_netif_t *esp_netif*, *esp_netif_dhcp_status_t *status*)

Get DHCP Server status.

Return

- ESP_OK

Parameters

- [in] `esp_netif`: Handle to esp-netif instance
- [out] `status`: If successful, the status of the DHCP server will be returned in this argument.

esp_err_t **esp_netif_dhcps_start** (*esp_netif_t *esp_netif*)

Start DHCP server (only if enabled in interface object)

Return

- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
- ESP_ERR_ESP_NETIF_DHCP_ALREADY_STARTED

Parameters

- [in] `esp_netif`: Handle to esp-netif instance

esp_err_t **esp_netif_dhcps_stop** (*esp_netif_t *esp_netif*)

Stop DHCP server (only if enabled in interface object)

Return

- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
- ESP_ERR_ESP_NETIF_DHCP_ALREADY_STOPPED
- ESP_ERR_ESP_NETIF_IF_NOT_READY

Parameters

- [in] `esp_netif`: Handle to esp-netif instance

esp_err_t **esp_netif_set_dns_info** (`esp_netif_t *esp_netif`, `esp_netif_dns_type_t type`, `esp_netif_dns_info_t *dns`)

Set DNS Server information.

This function behaves differently if DHCP server or client is enabled

If DHCP client is enabled, main and backup DNS servers will be updated automatically from the DHCP lease if the relevant DHCP options are set. Fallback DNS Server is never updated from the DHCP lease and is designed to be set via this API. If DHCP client is disabled, all DNS server types can be set via this API only.

If DHCP server is enabled, the Main DNS Server setting is used by the DHCP server to provide a DNS Server option to DHCP clients (Wi-Fi stations).

- The default Main DNS server is typically the IP of the Wi-Fi AP interface itself.
- This function can override it by setting server type ESP_NETIF_DNS_MAIN.
- Other DNS Server types are not supported for the Wi-Fi AP interface.

Return

- ESP_OK on success
- ESP_ERR_ESP_NETIF_INVALID_PARAMS invalid params

Parameters

- [in] `esp_netif`: Handle to esp-netif instance
- [in] `type`: Type of DNS Server to set: ESP_NETIF_DNS_MAIN, ESP_NETIF_DNS_BACKUP, ESP_NETIF_DNS_FALLBACK
- [in] `dns`: DNS Server address to set

esp_err_t **esp_netif_get_dns_info** (`esp_netif_t *esp_netif`, `esp_netif_dns_type_t type`, `esp_netif_dns_info_t *dns`)

Get DNS Server information.

Return the currently configured DNS Server address for the specified interface and Server type.

This may be result of a previous call to `esp_netif_set_dns_info()`. If the interface's DHCP client is enabled, the Main or Backup DNS Server may be set by the current DHCP lease.

Return

- ESP_OK on success
- ESP_ERR_ESP_NETIF_INVALID_PARAMS invalid params

Parameters

- [in] `esp_netif`: Handle to esp-netif instance
- [in] `type`: Type of DNS Server to get: ESP_NETIF_DNS_MAIN, ESP_NETIF_DNS_BACKUP, ESP_NETIF_DNS_FALLBACK
- [out] `dns`: DNS Server result is written here on success

esp_err_t **esp_netif_create_ip6_linklocal** (`esp_netif_t *esp_netif`)

Create interface link-local IPv6 address.

Cause the TCP/IP stack to create a link-local IPv6 address for the specified interface.

This function also registers a callback for the specified interface, so that if the link-local address becomes verified as the preferred address then a SYSTEM_EVENT_GOT_IP6 event will be sent.

Return

- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS

Parameters

- [in] `esp_netif`: Handle to esp-netif instance

esp_err_t **esp_netif_get_ip6_linklocal** (*esp_netif_t *esp_netif*, *esp_ip6_addr_t *if_ip6*)

Get interface link-local IPv6 address.

If the specified interface is up and a preferred link-local IPv6 address has been created for the interface, return a copy of it.

Return

- ESP_OK
- ESP_FAIL If interface is down, does not have a link-local IPv6 address, or the link-local IPv6 address is not a preferred address.

Parameters

- [in] *esp_netif*: Handle to esp-netif instance
- [out] *if_ip6*: IPv6 information will be returned in this argument if successful.

esp_err_t **esp_netif_get_ip6_global** (*esp_netif_t *esp_netif*, *esp_ip6_addr_t *if_ip6*)

Get interface global IPv6 address.

If the specified interface is up and a preferred global IPv6 address has been created for the interface, return a copy of it.

Return

- ESP_OK
- ESP_FAIL If interface is down, does not have a global IPv6 address, or the global IPv6 address is not a preferred address.

Parameters

- [in] *esp_netif*: Handle to esp-netif instance
- [out] *if_ip6*: IPv6 information will be returned in this argument if successful.

void **esp_netif_set_ip4_addr** (*esp_ip4_addr_t *addr*, *uint8_t a*, *uint8_t b*, *uint8_t c*, *uint8_t d*)

Sets IPv4 address to the specified octets.

Parameters

- [out] *addr*: IP address to be set
- *a*: the first octet (127 for IP 127.0.0.1)
- *b*:
- *c*:
- *d*:

char* **esp_ip4addr_ntoa** (**const** *esp_ip4_addr_t *addr*, *char *buf*, *int buflen*)

Converts numeric IP address into decimal dotted ASCII representation.

Return either pointer to *buf* which now holds the ASCII representation of *addr* or NULL if *buf* was too small

Parameters

- *addr*: ip address in network order to convert
- *buf*: target buffer where the string is stored
- *buflen*: length of *buf*

uint32_t **esp_ip4addr_aton** (**const** *char *addr*)

Ascii internet address interpretation routine The value returned is in network order.

Return ip address in network order

Parameters

- *addr*: IP address in ascii representation (e.g. "127.0.0.1")

esp_netif_iodriver_handle **esp_netif_get_io_driver** (*esp_netif_t *esp_netif*)

Gets media driver handle for this esp-netif instance.

Return opaque pointer of related IO driver

Parameters

- [in] *esp_netif*: Handle to esp-netif instance

*esp_netif_t ****esp_netif_get_handle_from_ifkey** (**const** *char *if_key*)

Searches over a list of created objects to find an instance with supplied if key.

Return Handle to esp-netif instance

Parameters

- `if_key`: Textual description of network interface

`esp_netif_flags_t esp_netif_get_flags` (`esp_netif_t *esp_netif`)
Returns configured flags for this interface.

Return Configuration flags

Parameters

- `[in] esp_netif`: Handle to esp-netif instance

`const char *esp_netif_get_ifkey` (`esp_netif_t *esp_netif`)
Returns configured interface key for this esp-netif instance.

Return Textual description of related interface

Parameters

- `[in] esp_netif`: Handle to esp-netif instance

`const char *esp_netif_get_desc` (`esp_netif_t *esp_netif`)
Returns configured interface type for this esp-netif instance.

Return Enumerated type of this interface, such as station, AP, ethernet

Parameters

- `[in] esp_netif`: Handle to esp-netif instance

`int32_t esp_netif_get_event_id` (`esp_netif_t *esp_netif`, `esp_netif_ip_event_type_t event_type`)
Returns configured event for this esp-netif instance and supplied event type.

Return specific event id which is configured to be raised if the interface lost or acquired IP address -1 if supplied event_type is not known

Parameters

- `[in] esp_netif`: Handle to esp-netif instance
- `event_type`: (either get or lost IP)

`esp_netif_t *esp_netif_next` (`esp_netif_t *esp_netif`)
Iterates over list of interfaces. Returns first netif if NULL given as parameter.

Return First netif from the list if supplied parameter is NULL, next one otherwise

Parameters

- `[in] esp_netif`: Handle to esp-netif instance

`size_t esp_netif_get_nr_of_ifs` (`void`)
Returns number of registered esp_netif objects.

Return Number of esp_netifs

WiFi default API reference

Header File

- [esp_wifi/include/esp_wifi_default.h](#)

Functions

`esp_err_t esp_netif_attach_wifi_station` (`esp_netif_t *esp_netif`)
Attaches wifi station interface to supplied netif.

Return

- `ESP_OK` on success
- `ESP_FAIL` if attach failed

Parameters

- `esp_netif`: instance to attach the wifi station to

`esp_err_t esp_netif_attach_wifi_ap` (`esp_netif_t *esp_netif`)
Attaches wifi soft AP interface to supplied netif.

Return

- ESP_OK on success
- ESP_FAIL if attach failed

Parameters

- `esp_netif`: instance to attach the wifi AP to

esp_err_t **esp_wifi_set_default_wifi_sta_handlers** (void)

Sets default wifi event handlers for STA interface.

Return

- ESP_OK on success, error returned from `esp_event_handler_register` if failed

esp_err_t **esp_wifi_set_default_wifi_ap_handlers** (void)

Sets default wifi event handlers for STA interface.

Return

- ESP_OK on success, error returned from `esp_event_handler_register` if failed

esp_err_t **esp_wifi_clear_default_wifi_driver_and_handlers** (void **esp_netif*)

Clears default wifi event handlers for supplied network interface.

Return

- ESP_OK on success, error returned from `esp_event_handler_register` if failed

Parameters

- `esp_netif`: instance of corresponding if object

esp_netif_t ***esp_netif_create_default_wifi_ap** (void)

Creates default WIFI AP. In case of any init error this API aborts.

Return pointer to esp-netif instance

esp_netif_t ***esp_netif_create_default_wifi_sta** (void)

Creates default WIFI STA. In case of any init error this API aborts.

Return pointer to esp-netif instance

esp_err_t **esp_netif_create_default_wifi_mesh_netifs** (*esp_netif_t* *esp_netif_t* ***p_netif_sta*,
esp_netif_t ***p_netif_ap*)

Creates default STA and AP network interfaces for esp-mesh.

Both netifs are almost identical to the default station and softAP, but with DHCP client and server disabled. Please note that the DHCP client is typically enabled only if the device is promoted to a root node.

Returns created interfaces which could be ignored setting parameters to NULL if an application code does not need to save the interface instances for further processing.

Return ESP_OK on success

Parameters

- [out] `p_netif_sta`: pointer where the resultant STA interface is saved (if non NULL)
- [out] `p_netif_ap`: pointer where the resultant AP interface is saved (if non NULL)

TCP/IP Adapter Migration Guide

TCP/IP Adapter is a network interface abstraction component used in IDF prior to v4.1. This page outlines migration from `tcpip_adapter` API to its successor [ESP-NETIF](#).

Updating network connection code

Network stack initialization Simply replace `tcpip_adapter_init()` with `esp_netif_init()`. Please note that the [ESP-NETIF](#) initialization API returns standard error code and the `esp_netif_deinit()` for un-initialization is available.

Also replace `#include "tcpip_adapter.h"` with `#include "esp_netif.h"`.

Network interface creation TCP/IP Adapter defined these three interfaces statically:

- WiFi Station
- WiFi Access Point
- Ethernet

Network interface instance shall be explicitly constructed for the *ESP-NETIF* to enable its connection to the TCP/IP stack. For example initialization code for WiFi has to explicitly call `esp_netif_create_default_wifi_sta()`; or `esp_netif_create_default_wifi_ap()`; after the TCP/IP stack and the event loop have been initialized. Please consult an example initialization code for these three interfaces:

- WiFi Station: [wifi/getting_started/station/main/station_example_main.c](#)
- WiFi Access Point: [wifi/getting_started/softAP/main/softap_example_main.c](#)
- Ethernet: [ethernet/basic/main/ethernet_example_main.c](#)

Replacing other tcpip_adapter API All the `tcpip_adapter` functions have their `esp-netif` counter-part. Please refer to the `esp_netif.h` grouped into these sections:

- [Setters/Getters](#)
- [DHCP](#)
- [DNS](#)
- [IP address](#)

Default event handlers Event handlers are moved from `tcpip_adapter` to appropriate driver code. There is no change from application code perspective, all events shall be handled in the same way. Please note that within IP related event handlers, application code usually receives IP addresses in a form of `esp-netif` specific struct (not the LwIP structs, but binary compatible). This is the preferred way of printing the address:

```
ESP_LOGI(TAG, "got ip:" IPSTR "\n", IP2STR(&event->ip_info.ip));
```

Instead of

```
ESP_LOGI(TAG, "got ip:%s\n", ip4addr_ntoa(&event->ip_info.ip));
```

Since `ip4addr_ntoa()` is a LwIP API, the `esp-netif` provides `esp_ip4addr_ntoa()` as a replacement, but the above method is generally preferred.

IP addresses It is preferred to use `esp-netif` defined IP structures. Please note that the LwIP structs will still work when default compatibility enabled. * [esp-netif IP address definitions](#)

Next steps Additional step in porting an application to fully benefit from the *ESP-NETIF* is to disable the `tcpip_adapter` compatibility layer in the component configuration: `ESP NETIF Adapter -> Enable backward compatible tcpip_adapter interface` and check if the project compiles. TCP/IP adapter brings many include dependencies and this step might help in decoupling the application from using specific TCP/IP stack API directly.

ESP-NETIF Custom I/O Driver

This section outlines implementing a new I/O driver with `esp-netif` connection capabilities. By convention the I/O driver has to register itself as an `esp-netif` driver and thus holds a dependency on `esp-netif` component and is responsible for providing data path functions, post-attach callback and in most cases also default event handlers to define network interface actions based on driver's lifecycle transitions.

Packet input/output As shown in the diagram, the following three API functions for the packet data path must be defined for connecting with esp-netif:

- `esp_netif_transmit()`
- `esp_netif_free_rx_buffer()`
- `esp_netif_receive()`

The first two functions for transmitting and freeing the rx buffer are provided as callbacks, i.e. they get called from esp-netif (and its underlying TCP/IP stack) and I/O driver provides their implementation.

The receiving function on the other hand gets called from the I/O driver, so that the driver's code simply calls `esp_netif_receive()` on a new data received event.

Post attach callback A final part of the network interface initialization consists of attaching the esp-netif instance to the I/O driver, by means of calling the following API:

```
esp_err_t esp_netif_attach(esp_netif_t *esp_netif, esp_netif_iodriver_handle_t
↳driver_handle);
```

It is assumed that the `esp_netif_iodriver_handle` is a pointer to driver's object, a struct derived from `struct esp_netif_driver_base_s`, so that the first member of I/O driver structure must be this base structure with pointers to

- post-attach function callback
- related esp-netif instance

As a consequence the I/O driver has to create an instance of the struct per below:

```
typedef struct my_netif_driver_s {
    esp_netif_driver_base_t base;           /*!< base structure reserved as_
↳esp-netif driver */
    driver_impl_t *h;                       /*!< handle of driver_
↳implementation */
} my_netif_driver_t;
```

with actual values of `my_netif_driver_t::base.post_attach` and the actual drivers handle `my_netif_driver_t::h`. So when the `esp_netif_attach()` gets called from the initialization code, the post-attach callback from I/O driver's code gets executed to mutually register callbacks between esp-netif and I/O driver instances. Typically the driver is started as well in the post-attach callback. An example of a simple post-attach callback is outlined below:

```
static esp_err_t my_post_attach_start(esp_netif_t * esp_netif, void * args)
{
    my_netif_driver_t *driver = args;
    const esp_netif_driver_ifconfig_t driver_ifconfig = {
        .driver_free_rx_buffer = my_free_rx_buf,
        .transmit = my_transmit,
        .handle = driver->driver_impl
    };
    driver->base.netif = esp_netif;
    ESP_ERROR_CHECK(esp_netif_set_driver_config(esp_netif, &driver_ifconfig));
    my_driver_start(driver->driver_impl);
    return ESP_OK;
}
```

Default handlers I/O drivers also typically provide default definitions of lifecycle behaviour of related network interfaces based on state transitions of I/O drivers. For example `driver start -> network start`, etc. An example of such a default handler is provided below:

```

esp_err_t my_driver_netif_set_default_handlers(my_netif_driver_t *driver, esp_
↔netif_t * esp_netif)
{
    driver_set_event_handler(driver->driver_impl, esp_netif_action_start, MY_DRV_
↔EVENT_START, esp_netif);
    driver_set_event_handler(driver->driver_impl, esp_netif_action_stop, MY_DRV_
↔EVENT_STOP, esp_netif);
    return ESP_OK;
}

```

Network stack connection The packet data path functions for transmitting and freeing the rx buffer (defined in the I/O driver) are called from the esp-netif, specifically from its TCP/IP stack connecting layer. The following API reference outlines these network stack interaction with the esp-netif.

Header File

- [esp_netif/include/esp_netif_net_stack.h](#)

Functions

`esp_netif_t *esp_netif_get_handle_from_netif_impl (void *dev)`

Returns esp-netif handle.

Return handle to related esp-netif instance

Parameters

- [in] dev: opaque ptr to network interface of specific TCP/IP stack

`void *esp_netif_get_netif_impl (esp_netif_t *esp_netif)`

Returns network stack specific implementation handle (if supported)

Note that it is not supported to acquire PPP netif impl pointer and this function will return NULL for esp_netif instances configured to PPP mode

Return handle to related network stack netif handle

Parameters

- [in] esp_netif: Handle to esp-netif instance

`esp_err_t esp_netif_transmit (esp_netif_t *esp_netif, void *data, size_t len)`

Outputs packets from the TCP/IP stack to the media to be transmitted.

This function gets called from network stack to output packets to IO driver.

Return ESP_OK on success, an error passed from the I/O driver otherwise

Parameters

- [in] esp_netif: Handle to esp-netif instance
- [in] data: Data to be transmitted
- [in] len: Length of the data frame

`void esp_netif_free_rx_buffer (void *esp_netif, void *buffer)`

Free the rx buffer allocated by the media driver.

This function gets called from network stack when the rx buffer to be freed in IO driver context, i.e. to deallocate a buffer owned by io driver (when data packets were passed to higher levels to avoid copying)

Parameters

- [in] esp_netif: Handle to esp-netif instance
- [in] buffer: Rx buffer pointer

Code examples for TCP/IP socket APIs are provided in the [protocols/sockets](#) directory of ESP-IDF examples.

The TCP/IP Adapter (legacy network interface library) has been deprecated, please consult the [TCP/IP Adapter Migration Guide](#) to update existing IDF applications.

2.2.4 Application Layer

Documentation for Application layer network protocols (above the IP Network layer) are provided in [Application Protocols](#).

2.3 Peripherals API

2.3.1 Analog to Digital Converter

Overview

The ESP32 integrates two 12-bit SAR ([Successive Approximation Register](#)) ADCs supporting a total of 18 measurement channels (analog enabled pins).

The ADC driver API supports ADC1 (8 channels, attached to GPIOs 32 - 39), and ADC2 (10 channels, attached to GPIOs 0, 2, 4, 12 - 15 and 25 - 27). However, the usage of ADC2 has some restrictions for the application:

1. ADC2 is used by the Wi-Fi driver. Therefore the application can only use ADC2 when the Wi-Fi driver has not started.
2. Some of the ADC2 pins are used as strapping pins (GPIO 0, 2, 15) thus cannot be used freely. Such is the case in the following official Development Kits:
 - [ESP32 DevKitC](#): GPIO 0 cannot be used due to external auto program circuits.
 - [ESP-WROVER-KIT](#): GPIO 0, 2, 4 and 15 cannot be used due to external connections for different purposes.

Configuration and Reading ADC

The ADC should be configured before reading is taken.

- For ADC1, configure desired precision and attenuation by calling functions [adc1_config_width\(\)](#) and [adc1_config_channel_atten\(\)](#).
- For ADC2, configure the attenuation by [adc2_config_channel_atten\(\)](#). The reading width of ADC2 is configured every time you take the reading.

Attenuation configuration is done per channel, see [adc1_channel_t](#) and [adc2_channel_t](#), set as a parameter of above functions.

Then it is possible to read ADC conversion result with [adc1_get_raw\(\)](#) and [adc2_get_raw\(\)](#). Reading width of ADC2 should be set as a parameter of [adc2_get_raw\(\)](#) instead of in the configuration functions.

Note: Since the ADC2 is shared with the WIFI module, which has higher priority, reading operation of [adc2_get_raw\(\)](#) will fail between [esp_wifi_start\(\)](#) and [esp_wifi_stop\(\)](#). Use the return code to see whether the reading is successful.

It is also possible to read the internal hall effect sensor via ADC1 by calling dedicated function [hall_sensor_read\(\)](#). Note that even the hall sensor is internal to ESP32, reading from it uses channels 0 and 3 of ADC1 (GPIO 36 and 39). Do not connect anything else to these pins and do not change their configuration. Otherwise it may affect the measurement of low value signal from the sensor.

This API provides convenient way to configure ADC1 for reading from [ULP](#). To do so, call function [adc1_ulp_enable\(\)](#) and then set precision and attenuation as discussed above.

There is another specific function [adc2_vref_to_gpio\(\)](#) used to route internal reference voltage to a GPIO pin. It comes handy to calibrate ADC reading and this is discussed in section [Minimizing Noise](#).

Application Examples

Reading voltage on ADC1 channel 0 (GPIO 36):

```
#include <driver/adc.h>

...

adc1_config_width(ADC_WIDTH_BIT_12);
adc1_config_channel_atten(ADC1_CHANNEL_0, ADC_ATTEN_DB_0);
int val = adc1_get_raw(ADC1_CHANNEL_0);
```

The input voltage in above example is from 0 to 1.1V (0 dB attenuation). The input range can be extended by setting higher attenuation, see [adc_atten_t](#). An example using the ADC driver including calibration (discussed below) is available in esp-idf: [peripherals/adc](#)

Reading voltage on ADC2 channel 7 (GPIO 27):

```
#include <driver/adc.h>

...

int read_raw;
adc2_config_channel_atten( ADC2_CHANNEL_7, ADC_ATTEN_0db );

esp_err_t r = adc2_get_raw( ADC2_CHANNEL_7, ADC_WIDTH_12Bit, &read_raw);
if ( r == ESP_OK ) {
    printf("%d\n", read_raw );
} else if ( r == ESP_ERR_TIMEOUT ) {
    printf("ADC2 used by Wi-Fi.\n");
}
```

The reading may fail due to collision with Wi-Fi, should check it. An example using the ADC2 driver to read the output of DAC is available in esp-idf: [peripherals/adc2](#)

Reading the internal hall effect sensor:

```
#include <driver/adc.h>

...

adc1_config_width(ADC_WIDTH_BIT_12);
int val = hall_sensor_read();
```

The value read in both these examples is 12 bits wide (range 0-4095).

Minimizing Noise

The ESP32 ADC can be sensitive to noise leading to large discrepancies in ADC readings. To minimize noise, users may connect a 0.1uF capacitor to the ADC input pad in use. Multisampling may also be used to further mitigate the effects of noise.

ADC Calibration

The [esp_adc_cal/include/esp_adc_cal.h](#) API provides functions to correct for differences in measured voltages caused by variation of ADC reference voltages (V_{ref}) between chips. Per design the ADC reference voltage is 1100mV, however the true reference voltage can range from 1000mV to 1200mV amongst different ESP32s.

Correcting ADC readings using this API involves characterizing one of the ADCs at a given attenuation to obtain a characteristics curve (ADC-Voltage curve) that takes into account the difference in ADC reference voltage. The characteristics curve is in the form of $y = \text{coeff_a} * x + \text{coeff_b}$ and is used to convert ADC readings to

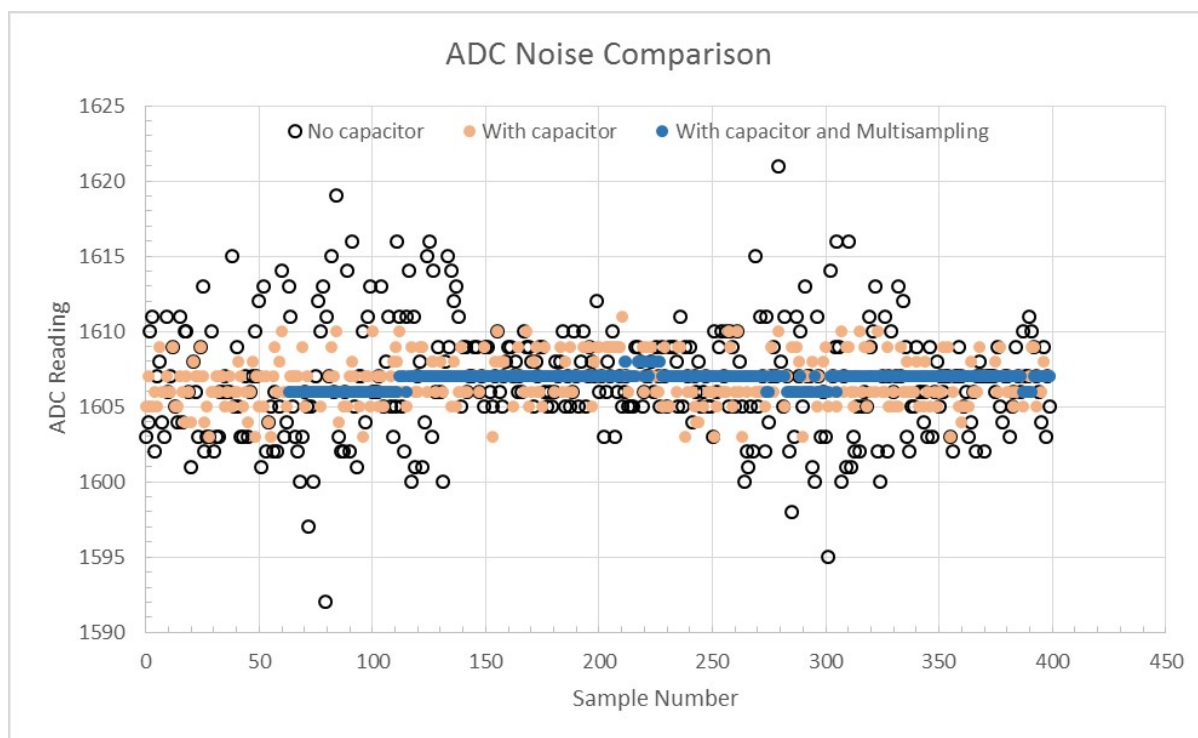


Fig. 4: Graph illustrating noise mitigation using capacitor and multisampling of 64 samples.

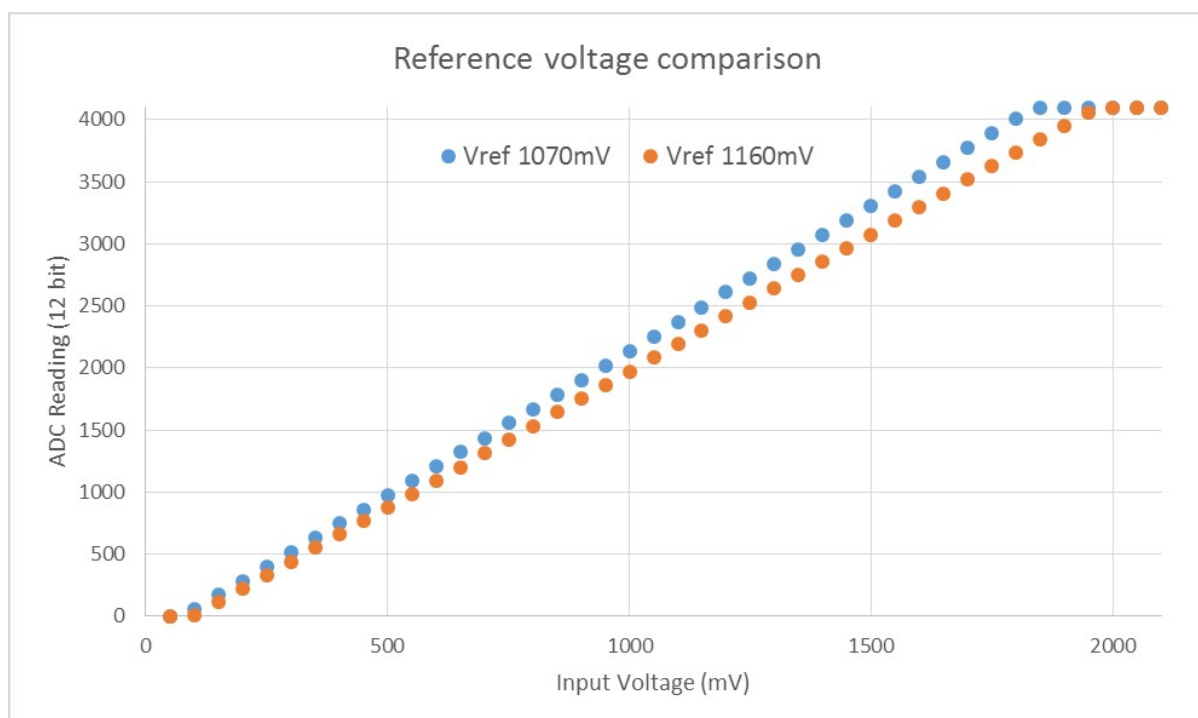


Fig. 5: Graph illustrating effect of differing reference voltages on the ADC voltage curve.

voltages in mV. Calculation of the characteristics curve is based on calibration values which can be stored in eFuse or provided by the user.

Calibration Values Calibration values are used to generate characteristic curves that account for the unique ADC reference voltage of a particular ESP32. There are currently three sources of calibration values. The availability of these calibration values will depend on the type and production date of the ESP32 chip/module.

- **Two Point** values represent each of the ADCs' readings at 150mV and 850mV. To obtain more accurate calibration results these values should be measured by user and burned into eFuse BLOCK3.
- **eFuse Vref** represents the true ADC reference voltage. This value is measured and burned into eFuse BLOCK0 during factory calibration.
- **Default Vref** is an estimate of the ADC reference voltage provided by the user as a parameter during characterization. If Two Point or eFuse Vref values are unavailable, **Default Vref** will be used.

Individual measurement and burning of the **eFuse Vref** has been applied to ESP32-D0WD and ESP32-D0WDQ6 chips produced on/after the 1st week of 2018. Such chips may be recognized by date codes on/later than 012018 (see Line 4 on figure below).

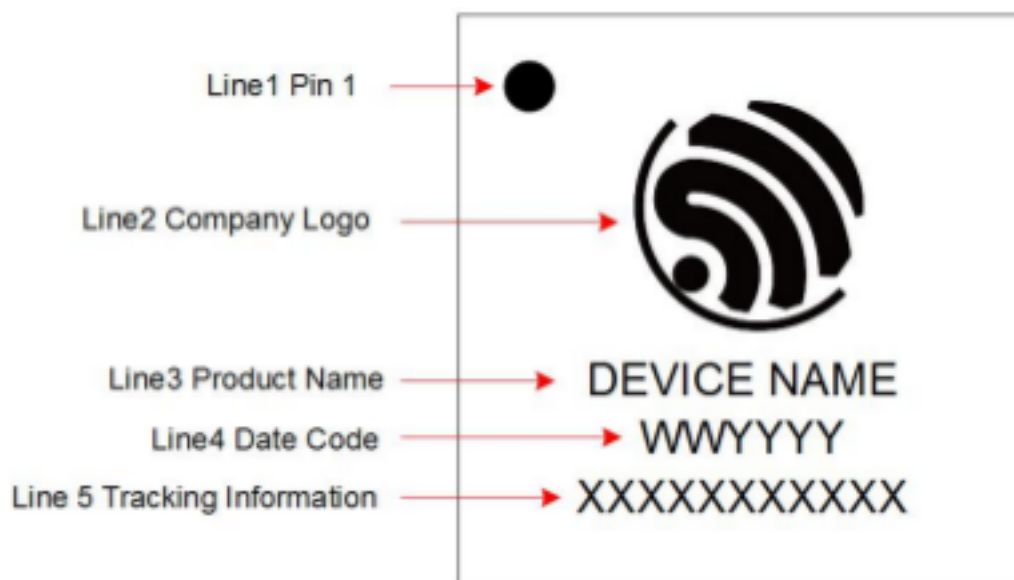


Fig. 6: ESP32 Chip Surface Marking

If you would like to purchase chips or modules with calibration, double check with distributor or Espressif directly.

If you are unable to check the date code (i.e. the chip may be enclosed inside a canned module, etc.), you can still verify if **eFuse Vref** is present by running [espfuse.py](#) tool with `adc_info` parameter

```
$IDF_PATH/components/esptool_py/esptool/espfuse.py --port /dev/ttyUSB0 adc_info
```

Replace `/dev/ttyUSB0` with ESP32 board's port name.

A chip that has specific **eFuse Vref** value programmed (in this case 1093mV) will be reported as follows:

```
ADC VRef calibration: 1093mV
```

In another example below the **eFuse Vref** is not programmed:

```
ADC VRef calibration: None (1100mV nominal)
```

For a chip with two point calibration the message will look similar to:

```

ADC Vref calibration: 1149mV
ADC readings stored in efuse BLK3:
  ADC1 Low reading (150mV): 306
  ADC1 High reading (850mV): 3153
  ADC2 Low reading (150mV): 389
  ADC2 High reading (850mV): 3206

```

Application Example For a full example see esp-idf: [peripherals/adc](#)

Characterizing an ADC at a particular attenuation:

```

#include "driver/adc.h"
#include "esp_adc_cal.h"

...

    //Characterize ADC at particular atten
    esp_adc_cal_characteristics_t *adc_chars = calloc(1, sizeof(esp_adc_cal_
↪characteristics_t));
    esp_adc_cal_value_t val_type = esp_adc_cal_characterize(unit, atten, ADC_WIDTH_
↪BIT_12, DEFAULT_VREF, adc_chars);
    //Check type of calibration value used to characterize ADC
    if (val_type == ESP_ADC_CAL_VAL_EFUSE_VREF) {
        printf("eFuse Vref");
    } else if (val_type == ESP_ADC_CAL_VAL_EFUSE_TP) {
        printf("Two Point");
    } else {
        printf("Default");
    }
}

```

Reading an ADC then converting the reading to a voltage:

```

#include "driver/adc.h"
#include "esp_adc_cal.h"

...

uint32_t reading = adc1_get_raw(ADC1_CHANNEL_5);
uint32_t voltage = esp_adc_cal_raw_to_voltage(reading, adc_chars);

```

Routing ADC reference voltage to GPIO, so it can be manually measured (for **Default Vref**):

```

#include "driver/adc.h"

...

esp_err_t status = adc2_vref_to_gpio(GPIO_NUM_25);
if (status == ESP_OK) {
    printf("v_ref routed to GPIO\n");
} else {
    printf("failed to route v_ref\n");
}

```

GPIO Lookup Macros

There are macros available to specify the GPIO number of a ADC channel, or vice versa. e.g.

1. ADC1_CHANNEL_0_GPIO_NUM is the GPIO number of ADC1 channel 0 (36);
2. ADC1_GPIO32_CHANNEL is the ADC1 channel number of GPIO 32 (ADC1 channel 4).

API Reference

This reference covers three components:

- [ADC driver](#)
- [ADC Calibration](#)
- [GPIO Lookup Macros](#)

ADC driver

Header File

- [driver/include/driver/adc.h](#)

Functions

esp_err_t **adc1_pad_get_io_num** (*adc1_channel_t* channel, *gpio_num_t* *gpio_num)

Get the GPIO number of a specific ADC1 channel.

Return

- ESP_OK if success
- ESP_ERR_INVALID_ARG if channel not valid

Parameters

- channel: Channel to get the GPIO number
- gpio_num: output buffer to hold the GPIO number

esp_err_t **adc1_config_width** (*adc_bits_width_t* width_bit)

Configure ADC1 capture width, meanwhile enable output invert for ADC1. The configuration is for all channels of ADC1.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- width_bit: Bit capture width for ADC1

esp_err_t **adc_set_data_width** (*adc_unit_t* adc_unit, *adc_bits_width_t* width_bit)

Configure ADC capture width.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- adc_unit: ADC unit index
- width_bit: Bit capture width for ADC unit.

esp_err_t **adc1_config_channel_atten** (*adc1_channel_t* channel, *adc_atten_t* atten)

Set the attenuation of a particular channel on ADC1, and configure its associated GPIO pin mux.

The default ADC full-scale voltage is 1.1 V. To read higher voltages (up to the pin maximum voltage, usually 3.3 V) requires setting >0 dB signal attenuation for that ADC channel.

Note For any given channel, this function must be called before the first time `adc1_get_raw()` is called for that channel.

Note This function can be called multiple times to configure multiple ADC channels simultaneously. `adc1_get_raw()` can then be called for any configured channel.

When VDD_A is 3.3 V:

- 0 dB attenuation (ADC_ATTEN_DB_0) gives full-scale voltage 1.1 V
- 2.5 dB attenuation (ADC_ATTEN_DB_2_5) gives full-scale voltage 1.5 V
- 6 dB attenuation (ADC_ATTEN_DB_6) gives full-scale voltage 2.2 V
- 11 dB attenuation (ADC_ATTEN_DB_11) gives full-scale voltage 3.9 V (see note below)

Due to ADC characteristics, most accurate results are obtained within the following approximate voltage ranges:

Note The full-scale voltage is the voltage corresponding to a maximum reading (depending on ADC1 configured bit width, this value is: 4095 for 12-bits, 2047 for 11-bits, 1023 for 10-bits, 511 for 9 bits.)

Note At 11 dB attenuation the maximum voltage is limited by VDD_A, not the full scale voltage.

- 0 dB attenuation (ADC_ATTEN_DB_0) between 100 and 950 mV
- 2.5 dB attenuation (ADC_ATTEN_DB_2_5) between 100 and 1250 mV
- 6 dB attenuation (ADC_ATTEN_DB_6) between 150 to 1750 mV
- 11 dB attenuation (ADC_ATTEN_DB_11) between 150 to 2450 mV

For maximum accuracy, use the ADC calibration APIs and measure voltages within these recommended ranges.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `channel`: ADC1 channel to configure
- `atten`: Attenuation level

int **adc1_get_raw** (*adc1_channel_t channel*)

Take an ADC1 reading from a single channel.

Note When the power switch of SARADC1, SARADC2, HALL sensor and AMP sensor is turned on, the input of GPIO36 and GPIO39 will be pulled down for about 80ns. When enabling power for any of these peripherals, ignore input from GPIO36 and GPIO39. Please refer to section 3.11 of ‘ECO_and_Workarounds_for_Bugs_in_ESP32’ for the description of this issue.

Note Call `adc1_config_width()` before the first time this function is called.

Note For any given channel, `adc1_config_channel_atten(channel)` must be called before the first time this function is called. Configuring a new channel does not prevent a previously configured channel from being read.

Return

- -1: Parameter error
- Other: ADC1 channel reading.

Parameters

- `channel`: ADC1 channel to read

void **adc_power_on** (void)

Enable ADC power.

void **adc_power_off** (void)

Power off SAR ADC.

void **adc_power_acquire** (void)

Increment the usage counter for ADC module. ADC will stay powered on while the counter is greater than 0. Call `adc_power_release` when done using the ADC.

void **adc_power_release** (void)

Decrement the usage counter for ADC module. ADC will stay powered on while the counter is greater than 0. Call this function when done using the ADC.

esp_err_t **adc_gpio_init** (*adc_unit_t adc_unit, adc_channel_t channel*)

Initialize ADC pad.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `adc_unit`: ADC unit index
- `channel`: ADC channel index

esp_err_t **adc_set_data_inv** (*adc_unit_t adc_unit, bool inv_en*)

Set ADC data invert.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `adc_unit`: ADC unit index
- `inv_en`: whether enable data invert

esp_err_t **adc_set_clk_div** (*uint8_t* *clk_div*)

Set ADC source clock.

Return

- ESP_OK success

Parameters

- `clk_div`: ADC clock divider, ADC clock is divided from APB clock

esp_err_t **adc_set_i2s_data_source** (*adc_i2s_source_t* *src*)

Set I2S data source.

Return

- ESP_OK success

Parameters

- `src`: I2S DMA data source, I2S DMA can get data from digital signals or from ADC.

esp_err_t **adc_i2s_mode_init** (*adc_unit_t* *adc_unit*, *adc_channel_t* *channel*)

Initialize I2S ADC mode.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `adc_unit`: ADC unit index
- `channel`: ADC channel index

void **adc1_ulp_enable** (void)

Configure ADC1 to be usable by the ULP.

This function reconfigures ADC1 to be controlled by the ULP. Effect of this function can be reverted using `adc1_get_raw` function.

Note that `adc1_config_channel_atten`, `adc1_config_width` functions need to be called to configure ADC1 channels, before ADC1 is used by the ULP.

int **hall_sensor_read** (void)

Read Hall Sensor.

Note When the power switch of SARADC1, SARADC2, HALL sensor and AMP sensor is turned on, the input of GPIO36 and GPIO39 will be pulled down for about 80ns. When enabling power for any of these peripherals, ignore input from GPIO36 and GPIO39. Please refer to section 3.11 of ‘ECO_and_Workarounds_for_Bugs_in_ESP32’ for the description of this issue.

Note The Hall Sensor uses channels 0 and 3 of ADC1. Do not configure these channels for use as ADC channels.

Note The ADC1 module must be enabled by calling `adc1_config_width()` before calling `hall_sensor_read()`. ADC1 should be configured for 12 bit readings, as the hall sensor readings are low values and do not cover the full range of the ADC.

Return The hall sensor reading.

esp_err_t **adc2_pad_get_io_num** (*adc2_channel_t* *channel*, *gpio_num_t* **gpio_num*)

Get the GPIO number of a specific ADC2 channel.

Return

- ESP_OK if success
- ESP_ERR_INVALID_ARG if channel not valid

Parameters

- `channel`: Channel to get the GPIO number
- `gpio_num`: output buffer to hold the GPIO number

esp_err_t **adc2_config_channel_atten** (*adc2_channel_t* channel, *adc_atten_t* atten)

Configure the ADC2 channel, including setting attenuation.

The default ADC full-scale voltage is 1.1 V. To read higher voltages (up to the pin maximum voltage, usually 3.3 V) requires setting >0 dB signal attenuation for that ADC channel.

Note This function also configures the input GPIO pin mux to connect it to the ADC2 channel. It must be called before calling `adc2_get_raw()` for this channel.

When VDD_A is 3.3 V:

- 0 dB attenuation (ADC_ATTEN_0db) gives full-scale voltage 1.1 V
- 2.5 dB attenuation (ADC_ATTEN_2_5db) gives full-scale voltage 1.5 V
- 6 dB attenuation (ADC_ATTEN_6db) gives full-scale voltage 2.2 V
- 11 dB attenuation (ADC_ATTEN_11db) gives full-scale voltage 3.9 V (see note below)

Note The full-scale voltage is the voltage corresponding to a maximum reading (depending on ADC2 configured bit width, this value is: 4095 for 12-bits, 2047 for 11-bits, 1023 for 10-bits, 511 for 9 bits.)

Note At 11 dB attenuation the maximum voltage is limited by VDD_A, not the full scale voltage.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- channel: ADC2 channel to configure
- atten: Attenuation level

esp_err_t **adc2_get_raw** (*adc2_channel_t* channel, *adc_bits_width_t* width_bit, int *raw_out)

Take an ADC2 reading on a single channel.

Note When the power switch of SARADC1, SARADC2, HALL sensor and AMP sensor is turned on, the input of GPIO36 and GPIO39 will be pulled down for about 80ns. When enabling power for any of these peripherals, ignore input from GPIO36 and GPIO39. Please refer to section 3.11 of ‘ECO_and_Workarounds_for_Bugs_in_ESP32’ for the description of this issue.

Note For a given channel, `adc2_config_channel_atten()` must be called before the first time this function is called. If Wi-Fi is started via `esp_wifi_start()`, this function will always fail with ESP_ERR_TIMEOUT.

Return

- ESP_OK if success
- ESP_ERR_TIMEOUT the WIFI is started, using the ADC2

Parameters

- channel: ADC2 channel to read
- width_bit: Bit capture width for ADC2
- raw_out: the variable to hold the output data.

esp_err_t **adc2_vref_to_gpio** (*gpio_num_t* gpio)

Output ADC2 reference voltage to GPIO 25 or 26 or 27.

This function utilizes the testing mux exclusive to ADC 2 to route the reference voltage one of ADC2's channels. Supported GPIOs are GPIOs 25, 26, and 27. This reference voltage can be manually read from the pin and used in the `esp_adc_cal` component.

Return

- ESP_OK: v_ref successfully routed to selected GPIO
- ESP_ERR_INVALID_ARG: Unsupported GPIO

Parameters

- [in] gpio: GPIO number (GPIOs 25, 26 and 27 are supported)

Macros

ADC_ATTEN_0db

ADC_ATTEN_2_5db

ADC_ATTEN_6db

ADC_ATTEN_11db
ADC_WIDTH_9Bit
ADC_WIDTH_10Bit
ADC_WIDTH_11Bit
ADC_WIDTH_12Bit

Enumerations

enum adc1_channel_t

Values:

ADC1_CHANNEL_0 = 0
ADC1 channel 0 is GPIO36 (ESP32), GPIO1 (ESP32-S2)

ADC1_CHANNEL_1
ADC1 channel 1 is GPIO37 (ESP32), GPIO2 (ESP32-S2)

ADC1_CHANNEL_2
ADC1 channel 2 is GPIO38 (ESP32), GPIO3 (ESP32-S2)

ADC1_CHANNEL_3
ADC1 channel 3 is GPIO39 (ESP32), GPIO4 (ESP32-S2)

ADC1_CHANNEL_4
ADC1 channel 4 is GPIO32 (ESP32), GPIO5 (ESP32-S2)

ADC1_CHANNEL_5
ADC1 channel 5 is GPIO33 (ESP32), GPIO6 (ESP32-S2)

ADC1_CHANNEL_6
ADC1 channel 6 is GPIO34 (ESP32), GPIO7 (ESP32-S2)

ADC1_CHANNEL_7
ADC1 channel 7 is GPIO35 (ESP32), GPIO8 (ESP32-S2)

enum adc2_channel_t

Values:

ADC2_CHANNEL_0 = 0
ADC2 channel 0 is GPIO4 (ESP32), GPIO11 (ESP32-S2)

ADC2_CHANNEL_1
ADC2 channel 1 is GPIO0 (ESP32), GPIO12 (ESP32-S2)

ADC2_CHANNEL_2
ADC2 channel 2 is GPIO2 (ESP32), GPIO13 (ESP32-S2)

ADC2_CHANNEL_3
ADC2 channel 3 is GPIO15 (ESP32), GPIO14 (ESP32-S2)

ADC2_CHANNEL_4
ADC2 channel 4 is GPIO13 (ESP32), GPIO15 (ESP32-S2)

ADC2_CHANNEL_5
ADC2 channel 5 is GPIO12 (ESP32), GPIO16 (ESP32-S2)

ADC2_CHANNEL_6
ADC2 channel 6 is GPIO14 (ESP32), GPIO17 (ESP32-S2)

ADC2_CHANNEL_7
ADC2 channel 7 is GPIO27 (ESP32), GPIO18 (ESP32-S2)

ADC2_CHANNEL_8
ADC2 channel 8 is GPIO25 (ESP32), GPIO19 (ESP32-S2)

ADC2_CHANNEL_9

ADC2 channel 9 is GPIO26 (ESP32), GPIO20 (ESP32-S2)

ADC2_CHANNEL_MAX**enum adc_unit_t***Values:***ADC_UNIT_1** = 1

SAR ADC 1

ADC_UNIT_2 = 2

SAR ADC 2, not supported yet

ADC_UNIT_BOTH = 3

SAR ADC 1 and 2, not supported yet

ADC_UNIT_ALTER = 7

SAR ADC 1 and 2 alternative mode, not supported yet

ADC_UNIT_MAX**enum adc_i2s_encode_t***Values:***ADC_ENCODE_12BIT**

ADC to I2S data format, [15:12]-channel [11:0]-12 bits ADC data

ADC_ENCODE_11BIT

ADC to I2S data format, [15]-1 [14:11]-channel [10:0]-11 bits ADC data

ADC_ENCODE_MAX**Header File**

- [soc/include/hal/adc_types.h](#)

Enumerations**enum adc_channel_t***Values:***ADC_CHANNEL_0** = 0

ADC channel

ADC_CHANNEL_1

ADC channel

ADC_CHANNEL_2

ADC channel

ADC_CHANNEL_3

ADC channel

ADC_CHANNEL_4

ADC channel

ADC_CHANNEL_5

ADC channel

ADC_CHANNEL_6

ADC channel

ADC_CHANNEL_7

ADC channel

ADC_CHANNEL_8

ADC channel

ADC_CHANNEL_9
ADC channel

ADC_CHANNEL_MAX

enum adc_atten_t

Values:

ADC_ATTEN_DB_0 = 0
The input voltage of ADC will be reduced to about 1/1

ADC_ATTEN_DB_2_5 = 1
The input voltage of ADC will be reduced to about 1/1.34

ADC_ATTEN_DB_6 = 2
The input voltage of ADC will be reduced to about 1/2

ADC_ATTEN_DB_11 = 3
The input voltage of ADC will be reduced to about 1/3.6

ADC_ATTEN_MAX

enum adc_i2s_source_t

Values:

ADC_I2S_DATA_SRC_IO_SIG = 0
I2S data from GPIO matrix signal

ADC_I2S_DATA_SRC_ADC = 1
I2S data from ADC

ADC_I2S_DATA_SRC_MAX

enum adc_bits_width_t

Values:

ADC_WIDTH_BIT_9 = 0
ADC capture width is 9Bit

ADC_WIDTH_BIT_10 = 1
ADC capture width is 10Bit

ADC_WIDTH_BIT_11 = 2
ADC capture width is 11Bit

ADC_WIDTH_BIT_12 = 3
ADC capture width is 12Bit

ADC_WIDTH_MAX

ADC Calibration

Header File

- [esp_adc_cal/include/esp_adc_cal.h](#)

Functions

esp_err_t **esp_adc_cal_check_efuse** (*esp_adc_cal_value_t* value_type)

Checks if ADC calibration values are burned into eFuse.

This function checks if ADC reference voltage or Two Point values have been burned to the eFuse of the current ESP32

Return

- **ESP_OK**: The calibration mode is supported in eFuse
- **ESP_ERR_NOT_SUPPORTED**: Error, eFuse values are not burned

- ESP_ERR_INVALID_ARG: Error, invalid argument (ESP_ADC_CAL_VAL_DEFAULT_VREF)

Parameters

- `value_type`: Type of calibration value (ESP_ADC_CAL_VAL_EFUSE_VREF or ESP_ADC_CAL_VAL_EFUSE_TP)

`esp_adc_cal_value_t` **esp_adc_cal_characterize** (`adc_unit_t` `adc_num`, `adc_atten_t` `atten`, `adc_bits_width_t` `bit_width`, `uint32_t` `default_vref`, `esp_adc_cal_characteristics_t` `*chars`)

Characterize an ADC at a particular attenuation.

This function will characterize the ADC at a particular attenuation and generate the ADC-Voltage curve in the form of $y = \text{coeff_a} * x + \text{coeff_b}$. Characterization can be based on Two Point values, eFuse Vref, or default Vref and the calibration values will be prioritized in that order.

Note Two Point values and eFuse Vref can be enabled/disabled using menuconfig.

Return

- ESP_ADC_CAL_VAL_EFUSE_VREF: eFuse Vref used for characterization
- ESP_ADC_CAL_VAL_EFUSE_TP: Two Point value used for characterization (only in Linear Mode)
- ESP_ADC_CAL_VAL_DEFAULT_VREF: Default Vref used for characterization

Parameters

- [in] `adc_num`: ADC to characterize (ADC_UNIT_1 or ADC_UNIT_2)
- [in] `atten`: Attenuation to characterize
- [in] `bit_width`: Bit width configuration of ADC
- [in] `default_vref`: Default ADC reference voltage in mV (used if eFuse values is not available)
- [out] `chars`: Pointer to empty structure used to store ADC characteristics

`uint32_t` **esp_adc_cal_raw_to_voltage** (`uint32_t` `adc_reading`, `esp_adc_cal_characteristics_t` `*chars`) **const**

Convert an ADC reading to voltage in mV.

This function converts an ADC reading to a voltage in mV based on the ADC's characteristics.

Note Characteristics structure must be initialized before this function is called (call `esp_adc_cal_characterize()`)

Return Voltage in mV

Parameters

- [in] `adc_reading`: ADC reading
- [in] `chars`: Pointer to initialized structure containing ADC characteristics

`esp_err_t` **esp_adc_cal_get_voltage** (`adc_channel_t` `channel`, `const` `esp_adc_cal_characteristics_t` `*chars`, `uint32_t` `*voltage`)

Reads an ADC and converts the reading to a voltage in mV.

This function reads an ADC then converts the raw reading to a voltage in mV based on the characteristics provided. The ADC that is read is also determined by the characteristics.

Note The Characteristics structure must be initialized before this function is called (call `esp_adc_cal_characterize()`)

Return

- ESP_OK: ADC read and converted to mV
- ESP_ERR_TIMEOUT: Error, timed out attempting to read ADC
- ESP_ERR_INVALID_ARG: Error due to invalid arguments

Parameters

- [in] `channel`: ADC Channel to read
- [in] `chars`: Pointer to initialized ADC characteristics structure
- [out] `voltage`: Pointer to store converted voltage

Structures

struct `esp_adc_cal_characteristics_t`

Structure storing characteristics of an ADC.

Note Call `esp_adc_cal_characterize()` to initialize the structure

Public Members

`adc_unit_t` **adc_num**

ADC number

`adc_atten_t` **atten**

ADC attenuation

`adc_bits_width_t` **bit_width**

ADC bit width

`uint32_t` **coeff_a**

Gradient of ADC-Voltage curve

`uint32_t` **coeff_b**

Offset of ADC-Voltage curve

`uint32_t` **vref**

Vref used by lookup table

const `uint32_t` ***low_curve**

Pointer to low Vref curve of lookup table (NULL if unused)

const `uint32_t` ***high_curve**

Pointer to high Vref curve of lookup table (NULL if unused)

Enumerations

enum `esp_adc_cal_value_t`

Type of calibration value used in characterization.

Values:

ESP_ADC_CAL_VAL_EFUSE_VREF = 0

Characterization based on reference voltage stored in eFuse

ESP_ADC_CAL_VAL_EFUSE_TP = 1

Characterization based on Two Point values stored in eFuse

ESP_ADC_CAL_VAL_DEFAULT_VREF = 2

Characterization based on default reference voltage

GPIO Lookup Macros

Header File

- [soc/esp32/include/soc/adc_channel.h](#)

Macros

ADC1_GPIO36_CHANNEL

ADC1_CHANNEL_0_GPIO_NUM

ADC1_GPIO37_CHANNEL

ADC1_CHANNEL_1_GPIO_NUM

ADC1_GPIO38_CHANNEL

ADC1_CHANNEL_2_GPIO_NUM

ADC1_GPIO39_CHANNEL

ADC1_CHANNEL_3_GPIO_NUM
ADC1_GPIO32_CHANNEL
ADC1_CHANNEL_4_GPIO_NUM
ADC1_GPIO33_CHANNEL
ADC1_CHANNEL_5_GPIO_NUM
ADC1_GPIO34_CHANNEL
ADC1_CHANNEL_6_GPIO_NUM
ADC1_GPIO35_CHANNEL
ADC1_CHANNEL_7_GPIO_NUM
ADC2_GPIO4_CHANNEL
ADC2_CHANNEL_0_GPIO_NUM
ADC2_GPIO0_CHANNEL
ADC2_CHANNEL_1_GPIO_NUM
ADC2_GPIO2_CHANNEL
ADC2_CHANNEL_2_GPIO_NUM
ADC2_GPIO15_CHANNEL
ADC2_CHANNEL_3_GPIO_NUM
ADC2_GPIO13_CHANNEL
ADC2_CHANNEL_4_GPIO_NUM
ADC2_GPIO12_CHANNEL
ADC2_CHANNEL_5_GPIO_NUM
ADC2_GPIO14_CHANNEL
ADC2_CHANNEL_6_GPIO_NUM
ADC2_GPIO27_CHANNEL
ADC2_CHANNEL_7_GPIO_NUM
ADC2_GPIO25_CHANNEL
ADC2_CHANNEL_8_GPIO_NUM
ADC2_GPIO26_CHANNEL
ADC2_CHANNEL_9_GPIO_NUM

2.3.2 Controller Area Network (CAN)

Overview

The ESP32's peripherals contains a CAN Controller that supports Standard Frame Format (11-bit ID) and Extended Frame Format (29-bit ID) of the CAN2.0B specification.

| |
|--|
| <p>Warning: The ESP32 CAN controller is not compatible with CAN FD frames and will interpret such frames as errors.</p> |
|--|

This programming guide is split into the following sections:

1. *Basic CAN Concepts*
2. *Signals Lines and Transceiver*
3. *Configuration*
4. *Driver Operation*
5. *Examples*

Basic CAN Concepts

Note: The following section only covers the basic aspects of CAN. For full details, see the CAN2.0B specification

The CAN protocol is a multi-master, multi-cast communication protocol with error detection/signalling and inbuilt message prioritization. The CAN protocol is commonly used as a communication bus in automotive applications.

Multi-master: Any node in a CAN bus is allowed initiate the transfer of data.

Multi-cast: When a node transmits a message, all nodes are able to receive the message (broadcast). However some nodes can selective choose which messages to accept via the use of acceptance filtering (multi-cast).

Error Detection and Signalling: Every CAN node will constantly monitor the CAN bus. When any node detects an error, it will signal the error by transmitting an error frame. Other nodes will receive the error frame and transmit their own error frames in response. This will result in an error detection being propagated to all nodes on the bus.

Message Priorities: If two nodes attempt to transmit simultaneously, the node transmitting the message with the lower ID will win arbitration. All other nodes will become receivers ensuring there is at most one transmitter at any time.

CAN Message Frames The CAN2.0B specification contains two frame formats known as **Extended Frame** and **Standard Frame** which contain 29-bit IDs and 11-bit IDs respectively. A CAN message consists of the following components

- 29-bit or 11-bit ID
- Data Length Code (DLC) between 0 to 8
- Up to 8 bytes of data (should match DLC)

Error States and Counters The CAN2.0B specification implements fault confinement by requiring every CAN node to maintain two internal error counters known as the **Transmit Error Counter (TEC)** and the **Receive Error Counter (REC)**. The two error counters are used to determine a CAN node's **error state**, and the counters are incremented and decremented following a set of rules (see CAN2.0B specification). These error states are known as **Error Active**, **Error Passive**, and **Bus-Off**.

Error Active: A CAN node is Error Active when **both TEC and REC are less than 128** and indicates a CAN node is operating normally. Error Active nodes are allowed to participate in CAN bus activities, and will actively signal any error conditions it detects by transmitting an **Active Error Flag** over the CAN bus.

Error Passive: A CAN node is Error Passive when **either the TEC or REC becomes greater than or equal to 128**. Error Passive nodes are still able to take part in CAN bus activities, but will instead transmit a **Passive Error Flag** upon detection of an error.

Bus-Off: A CAN node becomes Bus-Off when the **TEC becomes greater than or equal to 256**. A Bus-Off node is unable take part in CAN bus activity and will remain so until it undergoes bus recovery.

Signals Lines and Transceiver

The CAN controller does not contain a internal transceiver and therefore **requires an external transceiver** to operate. The type of external transceiver will depend on the application's physical layer specification (e.g. using SN65HVD23X transceivers for ISO 11898-2 compatibility).

The CAN controller's interface consists of 4 signal lines known as **TX**, **RX**, **BUS-OFF**, and **CLKOUT**. These four signal lines can be routed through the GPIO Matrix to GPIOs.

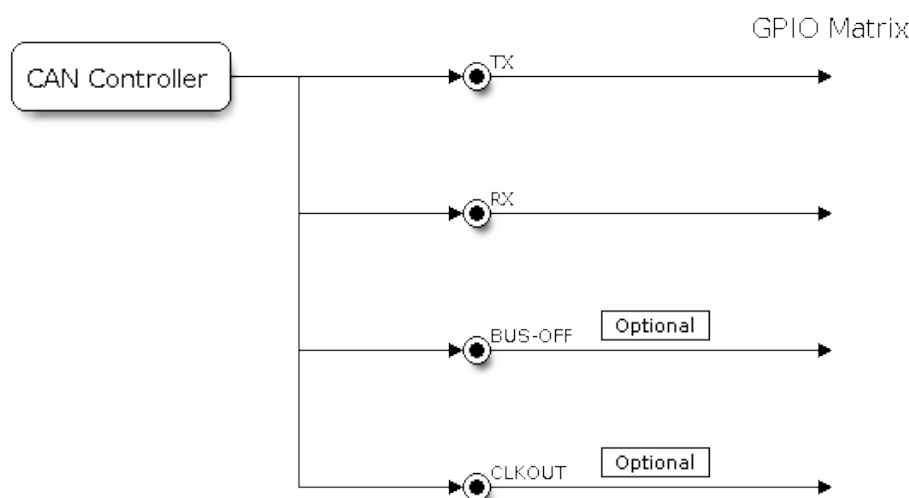


Fig. 7: Signal lines of the CAN controller

TX and RX: The TX and RX signal lines are required to interface with an external CAN transceiver. Both signal lines represent/interpret a dominant bit as a low logic level (0V), and a recessive bit as a high logic level (3.3V).

BUS-OFF: The BUS-OFF signal line is **optional** and is set to a low logic level (0V) whenever the CAN controller reaches a bus-off state. The BUS-OFF signal line is set to a high logic level (3.3V) otherwise.

CLKOUT: The CLKOUT signal line is **optional** and outputs a prescaled version of the CAN controller's source clock (APB Clock).

Note: An external transceiver **must internally tie the TX input and the RX output** such that a change in logic level to the TX signal line can be observed on the RX line. Failing to do so will cause the CAN controller to interpret differences in logic levels between the two signal lines as a lost in arbitration or a bit error.

Configuration

Operating Modes The CAN driver supports the following modes of operations:

Normal Mode: The normal operating mode allows the CAN controller to take part in bus activities such as transmitting and receiving messages/error frames. Acknowledgement from another CAN node is required when transmitting message frames.

No Ack Mode: The No Acknowledgement mode is similar to normal mode, however acknowledgements are not required when transmitting message frames. This mode is useful when self testing the CAN controller.

Listen Only Mode: This mode will prevent the CAN controller from taking part in bus activities. Therefore transmissions of messages/acknowledgement/error frames will be disabled. However the the CAN controller will still be able to receive messages (without acknowledging). This mode is suited for applications such as CAN bus monitoring.

Alerts The CAN driver contains an alert feature which is used to notify the application level of certain CAN driver events. Alerts are selectively enabled when the CAN driver is installed, but can be reconfigured during runtime by calling `can_reconfigure_alerts()`. The application can then wait for any enabled alerts to occur by calling `can_read_alerts()`. The CAN driver supports the following alerts:

| Alert | Description |
|--------------------------------|--|
| CAN_ALERT_TX_IDLE | No more messages queued for transmission |
| CAN_ALERT_TX_SUCCESS | The previous transmission was successful |
| CAN_ALERT_BELOW_ERR_WARN | Both error counters have dropped below error warning limit |
| CAN_ALERT_ERR_ACTIVE | CAN controller has become error active |
| CAN_ALERT_RECOVERY_IN_PROGRESS | CAN controller is undergoing bus recovery |
| CAN_ALERT_BUS_RECOVERED | CAN controller has successfully completed bus recovery |
| CAN_ALERT_ARB_LOST | The previous transmission lost arbitration |
| CAN_ALERT_ABOVE_ERR_WARN | One of the error counters have exceeded the error warning limit |
| CAN_ALERT_BUS_ERROR | A (Bit, Stuff, CRC, Form, ACK) error has occurred on the bus |
| CAN_ALERT_TX_FAILED | The previous transmission has failed |
| CAN_ALERT_RX_QUEUE_FULL | The RX queue is full causing a received frame to be lost |
| CAN_ALERT_ERR_PASS | CAN controller has become error passive |
| CAN_ALERT_BUS_OFF | Bus-off condition occurred. CAN controller can no longer influence bus |

Note: The **error warning limit** can be used to preemptively warn the application of bus errors before the error passive state is reached. By default the CAN driver sets the **error warning limit** to **96**. The CAN_ALERT_ABOVE_ERR_WARN is raised when the TEC or REC becomes larger then or equal to the error warning limit. The CAN_ALERT_BELOW_ERR_WARN is raised when both TEC and REC return back to values below **96**.

Note: When enabling alerts, the CAN_ALERT_AND_LOG flag can be used to cause the CAN driver to log any raised alerts to UART. However, alert logging is disabled and CAN_ALERT_AND_LOG if the *CONFIG_CAN_ISR_IN_IRAM* option is enabled (see *Placing ISR into IRAM*).

Note: The CAN_ALERT_ALL and CAN_ALERT_NONE macros can also be used to enable/disable all alerts during configuration/reconfiguration.

Bit Timing The operating bit rate of the CAN controller is configured using the *can_timing_config_t* structure. The period of each bit is made up of multiple **time quanta**, and the period of a **time quanta** is determined by a prescaled version of the CAN controller's source clock. A single bit contains the following segments in the following order:

1. The **Synchronization Segment** consists of a single time quanta
2. **Timing Segment 1** consists of 1 to 16 time quanta before sample point
3. **Timing Segment 2** consists of 1 to 8 time quanta after sample point

The **Baudrate Prescaler** is used to determine the period of each time quanta by dividing the CAN controller's source clock (80 MHz APB clock). The `brp` can be **any even number from 2 to 128**. If the ESP32 is a revision 2 or later chip, the `brp` will also support **any multiple of 4 from 132 to 256**, and can be enabled by setting the *CONFIG_ESP32_REV_MIN* to revision 2 or higher.

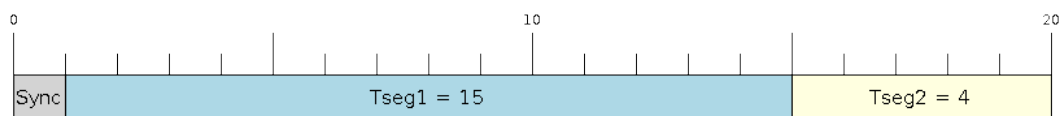


Fig. 8: Bit timing configuration for 500kbit/s given BRP = 8

The sample point of a bit is located on the intersection of Timing Segment 1 and 2. Enabling **Triple Sampling** will cause 3 time quanta to be sampled per bit instead of 1 (extra samples are located at the tail end of Timing Segment 1).

The **Synchronization Jump Width** is used to determine the maximum number of time quanta a single bit time can be lengthened/shortened for synchronization purposes. `sjw` can **range from 1 to 4**.

Note: Multiple combinations of `brp`, `tseg_1`, `tseg_2`, and `sjw` can achieve the same bit rate. Users should tune these values to the physical characteristics of their CAN bus by taking into account factors such as **propagation delay, node information processing time, and phase errors**.

Bit timing **macro initializers** are also available for commonly used CAN bus bit rates. The following macro initializers are provided by the CAN driver.

- `CAN_TIMING_CONFIG_12_5KBITS()`
- `CAN_TIMING_CONFIG_16KBITS()`
- `CAN_TIMING_CONFIG_20KBITS()`
- `CAN_TIMING_CONFIG_25KBITS()`
- `CAN_TIMING_CONFIG_50KBITS()`
- `CAN_TIMING_CONFIG_100KBITS()`
- `CAN_TIMING_CONFIG_125KBITS()`
- `CAN_TIMING_CONFIG_250KBITS()`
- `CAN_TIMING_CONFIG_500KBITS()`
- `CAN_TIMING_CONFIG_800KBITS()`
- `CAN_TIMING_CONFIG_1MBITS()`

Note: The macro initializers for 12.5K, 16K, and 20K bit rates are only available for ESP32 revision 2 or later.

Acceptance Filter The CAN controller contains a hardware acceptance filter which can be used to filter CAN messages of a particular ID. A node that filters out a message **will not receive the message, but will still acknowledge it**. Acceptance filters can make a node more efficient by filtering out messages sent over the CAN bus that are irrelevant to the CAN node in question. The CAN controller's acceptance filter is configured using two 32-bit values within `can_filter_config_t` known as the **acceptance code** and the **acceptance mask**.

The **acceptance code** specifies the bit sequence which a message's ID, RTR, and data bytes must match in order for the message to be received by the CAN controller. The **acceptance mask** is a bit sequence specifying which bits of the acceptance code can be ignored. This allows for messages of different IDs to be accepted by a single acceptance code.

The acceptance filter can be used under **Single or Dual Filter Mode**. Single Filter Mode will use the acceptance code and mask to define a single filter. This allows for the first two data bytes of a standard frame to be filtered, or the entirety of an extended frame's 29-bit ID. The following diagram illustrates how the 32-bit acceptance code and mask will be interpreted under Single Filter Mode (Note: The yellow and blue fields represent standard and extended CAN frames respectively).

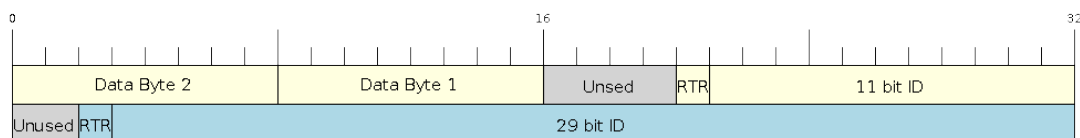


Fig. 9: Bit layout of single filter mode (Right side MSBit)

Dual Filter Mode will use the acceptance code and mask to define two separate filters allowing for increased flexibility of ID's to accept, but does not allow for all 29-bits of an extended ID to be filtered. The following diagram

illustrates how the 32-bit acceptance code and mask will be interpreted under **Dual Filter Mode** (Note: The yellow and blue fields represent standard and extended CAN frames respectively).

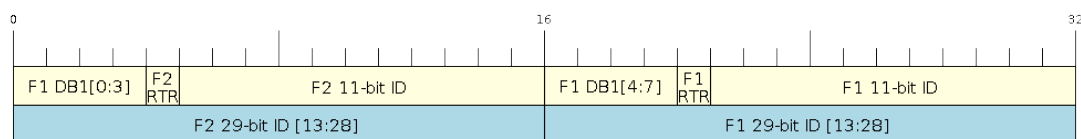


Fig. 10: Bit layout of dual filter mode (Right side MSBit)

Disabling TX Queue The TX queue can be disabled during configuration by setting the `tx_queue_len` member of `can_general_config_t` to 0. This will allow applications that do not require message transmission to save a small amount of memory when using the CAN driver.

Placing ISR into IRAM The CAN driver's ISR (Interrupt Service Routine) can be placed into IRAM so that the ISR can still run whilst the cache is disabled. Placing the ISR into IRAM may be necessary to maintain the CAN driver's functionality during lengthy cache disabling operations (such as SPI Flash writes, OTA updates etc). Whilst the cache is disabled, the ISR will continue to:

- Read received messages from the RX buffer and place them into the driver's RX queue.
- Load messages pending transmission from the driver's TX queue and write them into the TX buffer.

To place the CAN driver's ISR, users must do the following:

- Enable the `CONFIG_CAN_ISR_IN_IRAM` option using `idf.py menuconfig`.
- When calling `can_driver_install()`, the `intr_flags` member of `can_general_config_t` should set the `ESP_INTR_FLAG_IRAM` set.

Note: When the `CONFIG_CAN_ISR_IN_IRAM` option is enabled, the CAN driver will no longer log any alerts (i.e., the `CAN_ALERT_AND_LOG` flag will not have any effect).

Driver Operation

The CAN driver is designed with distinct states and strict rules regarding the functions or conditions that trigger a state transition. The following diagram illustrates the various states and their transitions.

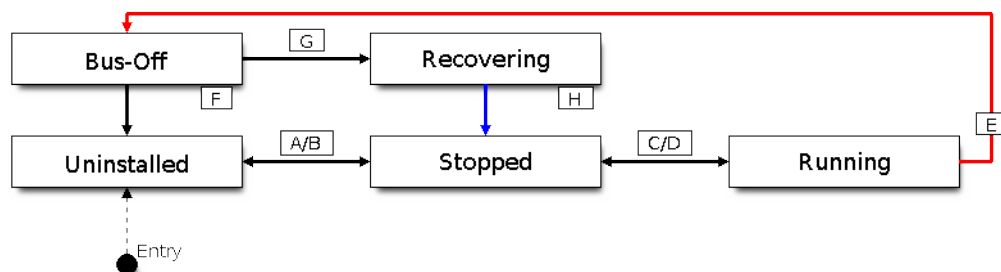


Fig. 11: State transition diagram of the CAN driver (see table below)

| Label | Transition | Action/Condition |
|-------|------------------------|--------------------------------------|
| A | Uninstalled -> Stopped | <code>can_driver_install()</code> |
| B | Stopped -> Uninstalled | <code>can_driver_uninstall()</code> |
| C | Stopped -> Running | <code>can_start()</code> |
| D | Running -> Stopped | <code>can_stop()</code> |
| E | Running -> Bus-Off | Transmit Error Counter \geq 256 |
| F | Bus-Off -> Uninstalled | <code>can_driver_uninstall()</code> |
| G | Bus-Off -> Recovering | <code>can_initiate_recovery()</code> |
| H | Recovering -> Stopped | 128 occurrences of bus-free signal |

Driver States Uninstalled: In the uninstalled state, no memory is allocated for the driver and the CAN controller is powered OFF.

Stopped: In this state, the CAN controller is powered ON and the CAN driver has been installed. However the CAN controller will be unable to take part in any CAN bus activities such as transmitting, receiving, or acknowledging messages.

Running: In the running state, the CAN controller is able to take part in bus activities. Therefore messages can be transmitted/received/acknowledged. Furthermore the CAN controller will be able to transmit error frames upon detection of errors on the CAN bus.

Bus-Off: The bus-off state is automatically entered when the CAN controller's Transmit Error Counter becomes greater than or equal to 256 (see CAN2.0B specification regarding error counter rules). The bus-off state indicates the occurrence of severe errors on the CAN bus or in the CAN controller. Whilst in the bus-off state, the CAN controller will be unable to take part in any CAN bus activities. To exit the bus-off state, the CAN controller must undergo the bus recovery process.

Recovering: The recovering state is entered when the CAN driver undergoes bus recovery. The CAN driver/controller will remain in the recovering state until the 128 occurrences of the bus-free signal (see CAN2.0B specification) is observed on the CAN bus.

Message Flags The CAN driver distinguishes different types of CAN messages by using the various bit field members of the `can_message_t` structure. These bit field members help distinguish whether a message is in standard or extended format, an RTR, and the type of transmission to use when transmitting such a message. These bit field members can also be toggled using the `flags` member of `can_message_t` and the following message flags:

| Flag | Description |
|------------------------------------|---|
| <code>CAN_MSG_FLAG_EXTENDED</code> | Message is in Extended Frame Format (29bit ID) |
| <code>CAN_MSG_FLAG_RTR</code> | Message is a Remote Transmit Request |
| <code>CAN_MSG_FLAG_SS</code> | Transmit message using Single Shot Transmission (Message will not be retransmitted upon error or loss of arbitration). Unused for received message. |
| <code>CAN_MSG_FLAG_SELF</code> | Transmit message using Self Reception Request (Transmitted message will also received by the same node). Unused for received message. |
| <code>CAN_MSG_FLAG_DLC</code> | Message ID Data length code is larger than 8. This will break compliance with CAN2.0B |
| <code>CAN_MSG_FLAG_NONE</code> | Clears all bit fields. Equivalent to a Standard Frame Format (11bit ID) Data Frame. |

Examples

Configuration & Installation The following code snippet demonstrates how to configure, install, and start the CAN driver via the use of the various configuration structures, macro initializers, the `can_driver_install()` function, and the `can_start()` function.

```
#include "driver/gpio.h"
#include "driver/can.h"

void app_main()
```

(continues on next page)

(continued from previous page)

```

{
    //Initialize configuration structures using macro initializers
    can_general_config_t g_config = CAN_GENERAL_CONFIG_DEFAULT(GPIO_NUM_21, GPIO_
    ↪NUM_22, CAN_MODE_NORMAL);
    can_timing_config_t t_config = CAN_TIMING_CONFIG_500KBITS();
    can_filter_config_t f_config = CAN_FILTER_CONFIG_ACCEPT_ALL();

    //Install CAN driver
    if (can_driver_install(&g_config, &t_config, &f_config) == ESP_OK) {
        printf("Driver installed\n");
    } else {
        printf("Failed to install driver\n");
        return;
    }

    //Start CAN driver
    if (can_start() == ESP_OK) {
        printf("Driver started\n");
    } else {
        printf("Failed to start driver\n");
        return;
    }

    ...
}

```

The usage of macro initializers are not mandatory and each of the configuration structures can be manually.

Message Transmission The following code snippet demonstrates how to transmit a message via the usage of the `can_message_t` type and `can_transmit()` function.

```

#include "driver/can.h"

...

//Configure message to transmit
can_message_t message;
message.identifier = 0xAAAA;
message.extd = 1;
message.data_length_code = 4;
for (int i = 0; i < 4; i++) {
    message.data[i] = 0;
}

//Queue message for transmission
if (can_transmit(&message, pdMS_TO_TICKS(1000)) == ESP_OK) {
    printf("Message queued for transmission\n");
} else {
    printf("Failed to queue message for transmission\n");
}

```

Message Reception The following code snippet demonstrates how to receive a message via the usage of the `can_message_t` type and `can_receive()` function.

```

#include "driver/can.h"

...

```

(continues on next page)

(continued from previous page)

```

//Wait for message to be received
can_message_t message;
if (can_receive(&message, pdMS_TO_TICKS(10000)) == ESP_OK) {
    printf("Message received\n");
} else {
    printf("Failed to receive message\n");
    return;
}

//Process received message
if (message.extd) {
    printf("Message is in Extended Format\n");
} else {
    printf("Message is in Standard Format\n");
}
printf("ID is %d\n", message.identifier);
if (!(message.rtr)) {
    for (int i = 0; i < message.data_length_code; i++) {
        printf("Data byte %d = %d\n", i, message.data[i]);
    }
}

```

Reconfiguring and Reading Alerts The following code snippet demonstrates how to reconfigure and read CAN driver alerts via the use of the `can_reconfigure_alerts()` and `can_read_alerts()` functions.

```

#include "driver/can.h"

...

//Reconfigure alerts to detect Error Passive and Bus-Off error states
uint32_t alerts_to_enable = CAN_ALERT_ERR_PASS | CAN_ALERT_BUS_OFF;
if (can_reconfigure_alerts(alerts_to_enable, NULL) == ESP_OK) {
    printf("Alerts reconfigured\n");
} else {
    printf("Failed to reconfigure alerts");
}

//Block indefinitely until an alert occurs
uint32_t alerts_triggered;
can_read_alerts(&alerts_triggered, portMAX_DELAY);

```

Stop and Uninstall The following code demonstrates how to stop and uninstall the CAN driver via the use of the `can_stop()` and `can_driver_uninstall()` functions.

```

#include "driver/can.h"

...

//Stop the CAN driver
if (can_stop() == ESP_OK) {
    printf("Driver stopped\n");
} else {
    printf("Failed to stop driver\n");
    return;
}

//Uninstall the CAN driver
if (can_driver_uninstall() == ESP_OK) {

```

(continues on next page)

(continued from previous page)

```

    printf("Driver uninstalled\n");
} else {
    printf("Failed to uninstall driver\n");
    return;
}

```

Multiple ID Filter Configuration The acceptance mask in `can_filter_config_t` can be configured such that two or more IDs will be accepted for a single filter. For a particular filter to accept multiple IDs, the conflicting bit positions amongst the IDs must be set in the acceptance mask. The acceptance code can be set to any one of the IDs.

The following example shows how to calculate the acceptance mask given multiple IDs:

```

ID1 = 11'b101 1010 0000
ID2 = 11'b101 1010 0001
ID3 = 11'b101 1010 0100
ID4 = 11'b101 1010 1000
//Acceptance Mask
MASK = 11'b000 0000 1101

```

Application Examples **Network Example:** The CAN Network example demonstrates communication between two ESP32s using the CAN driver API. One CAN node acts as a network master initiate and ceasing the transfer of a data from another CAN node acting as a network slave. The example can be found via [peripherals/can/can_network](#).

Alert and Recovery Example: This example demonstrates how to use the CAN driver's alert and bus recovery API. The example purposely introduces errors on the CAN bus to put the CAN controller into the Bus-Off state. An alert is used to detect the Bus-Off state and trigger the bus recovery process. The example can be found via [peripherals/can/can_alert_and_recovery](#).

Self Test Example: This example uses the No Acknowledge Mode and Self Reception Request to cause the CAN controller to send and simultaneously receive a series of messages. This example can be used to verify if the connections between the CAN controller and the external transceiver are working correctly. The example can be found via [peripherals/can/can_self_test](#).

API Reference

Header File

- `soc/include/hal/can_types.h`

Structures

`struct can_message_t`

Structure to store a CAN message.

Note

Note The flags member is deprecated

Public Members

`uint32_t extd` : 1

Extended Frame Format (29bit ID)

`uint32_t rtr` : 1

Message is a Remote Transmit Request

`uint32_t ss` : 1

Transmit as a Single Shot Transmission. Unused for received.

uint32_t **self** : 1
Transmit as a Self Reception Request. Unused for received.

uint32_t **dlc_non_comp** : 1
Message's Data length code is larger than 8. This will break compliance with CAN2.0B.

uint32_t **reserved** : 27
Reserved bits

uint32_t **flags**
Alternate way to set message flags using message flag macros (see documentation)

uint32_t **identifier**
11 or 29 bit identifier

uint8_t **data_length_code**
Data length code

uint8_t **data**[CAN_FRAME_MAX_DLC]
Data bytes (not relevant in RTR frame)

struct can_timing_config_t
Structure for bit timing configuration of the CAN driver.

Note Macro initializers are available for this structure

Public Members

uint32_t **brp**
Baudrate prescaler (i.e., APB clock divider) can be any even number from 2 to 128. For ESP32 Rev 2 or later, multiples of 4 from 132 to 256 are also supported

uint8_t **tseg_1**
Timing segment 1 (Number of time quanta, between 1 to 16)

uint8_t **tseg_2**
Timing segment 2 (Number of time quanta, 1 to 8)

uint8_t **sjw**
Synchronization Jump Width (Max time quanta jump for synchronize from 1 to 4)

bool **triple_sampling**
Enables triple sampling when the CAN controller samples a bit

struct can_filter_config_t
Structure for acceptance filter configuration of the CAN driver (see documentation)

Note Macro initializers are available for this structure

Public Members

uint32_t **acceptance_code**
32-bit acceptance code

uint32_t **acceptance_mask**
32-bit acceptance mask

bool **single_filter**
Use Single Filter Mode (see documentation)

Macros

CAN_EXTD_ID_MASK
CAN2.0B Constants.

Bit mask for 29 bit Extended Frame Format ID

CAN_STD_ID_MASK

Bit mask for 11 bit Standard Frame Format ID

CAN_FRAME_MAX_DLC

Max data bytes allowed in CAN2.0

CAN_FRAME_EXTD_ID_LEN_BYTES

EFF ID requires 4 bytes (29bit)

CAN_FRAME_STD_ID_LEN_BYTES

SFF ID requires 2 bytes (11bit)

CAN_ERR_PASS_THRESH

Error counter threshold for error passive

Enumerations**enum can_mode_t**

CAN Controller operating modes.

Values:

CAN_MODE_NORMAL

Normal operating mode where CAN controller can send/receive/acknowledge messages

CAN_MODE_NO_ACK

Transmission does not require acknowledgment. Use this mode for self testing

CAN_MODE_LISTEN_ONLY

The CAN controller will not influence the bus (No transmissions or acknowledgments) but can receive messages

Header File

- [driver/include/driver/can.h](#)

Functions

esp_err_t **can_driver_install** (**const** *can_general_config_t* *g_config, **const** *can_timing_config_t* *t_config, **const** *can_filter_config_t* *f_config)

Install CAN driver.

This function installs the CAN driver using three configuration structures. The required memory is allocated and the CAN driver is placed in the stopped state after running this function.

Note Macro initializers are available for the configuration structures (see documentation)

Note To reinstall the CAN driver, call `can_driver_uninstall()` first

Return

- ESP_OK: Successfully installed CAN driver
- ESP_ERR_INVALID_ARG: Arguments are invalid
- ESP_ERR_NO_MEM: Insufficient memory
- ESP_ERR_INVALID_STATE: Driver is already installed

Parameters

- [in] g_config: General configuration structure
- [in] t_config: Timing configuration structure
- [in] f_config: Filter configuration structure

esp_err_t **can_driver_uninstall** (void)

Uninstall the CAN driver.

This function uninstalls the CAN driver, freeing the memory utilized by the driver. This function can only be called when the driver is in the stopped state or the bus-off state.

Warning The application must ensure that no tasks are blocked on TX/RX queues or alerts when this function is called.

Return

- ESP_OK: Successfully uninstalled CAN driver
- ESP_ERR_INVALID_STATE: Driver is not in stopped/bus-off state, or is not installed

esp_err_t **can_start** (void)

Start the CAN driver.

This function starts the CAN driver, putting the CAN driver into the running state. This allows the CAN driver to participate in CAN bus activities such as transmitting/receiving messages. The RX queue is reset in this function, clearing any unread messages. This function can only be called when the CAN driver is in the stopped state.

Return

- ESP_OK: CAN driver is now running
- ESP_ERR_INVALID_STATE: Driver is not in stopped state, or is not installed

esp_err_t **can_stop** (void)

Stop the CAN driver.

This function stops the CAN driver, preventing any further message from being transmitted or received until `can_start()` is called. Any messages in the TX queue are cleared. Any messages in the RX queue should be read by the application after this function is called. This function can only be called when the CAN driver is in the running state.

Warning A message currently being transmitted/received on the CAN bus will be ceased immediately. This may lead to other CAN nodes interpreting the unfinished message as an error.

Return

- ESP_OK: CAN driver is now Stopped
- ESP_ERR_INVALID_STATE: Driver is not in running state, or is not installed

esp_err_t **can_transmit** (**const** *can_message_t* *message, TickType_t ticks_to_wait)

Transmit a CAN message.

This function queues a CAN message for transmission. Transmission will start immediately if no other messages are queued for transmission. If the TX queue is full, this function will block until more space becomes available or until it timesout. If the TX queue is disabled (TX queue length = 0 in configuration), this function will return immediately if another message is undergoing transmission. This function can only be called when the CAN driver is in the running state and cannot be called under Listen Only Mode.

Note This function does not guarantee that the transmission is successful. The TX_SUCCESS/TX_FAILED alert can be enabled to alert the application upon the success/failure of a transmission.

Note The TX_IDLE alert can be used to alert the application when no other messages are awaiting transmission.

Return

- ESP_OK: Transmission successfully queued/initiated
- ESP_ERR_INVALID_ARG: Arguments are invalid
- ESP_ERR_TIMEOUT: Timed out waiting for space on TX queue
- ESP_FAIL: TX queue is disabled and another message is currently transmitting
- ESP_ERR_INVALID_STATE: CAN driver is not in running state, or is not installed
- ESP_ERR_NOT_SUPPORTED: Listen Only Mode does not support transmissions

Parameters

- [in] message: Message to transmit
- [in] ticks_to_wait: Number of FreeRTOS ticks to block on the TX queue

esp_err_t **can_receive** (*can_message_t* *message, TickType_t ticks_to_wait)

Receive a CAN message.

This function receives a message from the RX queue. The flags field of the message structure will indicate the type of message received. This function will block if there are no messages in the RX queue

Warning The flags field of the received message should be checked to determine if the received message contains any data bytes.

Return

- ESP_OK: Message successfully received from RX queue
- ESP_ERR_TIMEOUT: Timed out waiting for message

- `ESP_ERR_INVALID_ARG`: Arguments are invalid
- `ESP_ERR_INVALID_STATE`: CAN driver is not installed

Parameters

- [out] `message`: Received message
- [in] `ticks_to_wait`: Number of FreeRTOS ticks to block on RX queue

esp_err_t **can_read_alerts** (`uint32_t *alerts`, `TickType_t ticks_to_wait`)

Read CAN driver alerts.

This function will read the alerts raised by the CAN driver. If no alert has been when this function is called, this function will block until an alert occurs or until it timeouts.

Note Multiple alerts can be raised simultaneously. The application should check for all alerts that have been enabled.

Return

- `ESP_OK`: Alerts read
- `ESP_ERR_TIMEOUT`: Timed out waiting for alerts
- `ESP_ERR_INVALID_ARG`: Arguments are invalid
- `ESP_ERR_INVALID_STATE`: CAN driver is not installed

Parameters

- [out] `alerts`: Bit field of raised alerts (see documentation for alert flags)
- [in] `ticks_to_wait`: Number of FreeRTOS ticks to block for alert

esp_err_t **can_reconfigure_alerts** (`uint32_t alerts_enabled`, `uint32_t *current_alerts`)

Reconfigure which alerts are enabled.

This function reconfigures which alerts are enabled. If there are alerts which have not been read whilst reconfiguring, this function can read those alerts.

Return

- `ESP_OK`: Alerts reconfigured
- `ESP_ERR_INVALID_STATE`: CAN driver is not installed

Parameters

- [in] `alerts_enabled`: Bit field of alerts to enable (see documentation for alert flags)
- [out] `current_alerts`: Bit field of currently raised alerts. Set to NULL if unused

esp_err_t **can_initiate_recovery** (`void`)

Start the bus recovery process.

This function initiates the bus recovery process when the CAN driver is in the bus-off state. Once initiated, the CAN driver will enter the recovering state and wait for 128 occurrences of the bus-free signal on the CAN bus before returning to the stopped state. This function will reset the TX queue, clearing any messages pending transmission.

Note The `BUS_RECOVERED` alert can be enabled to alert the application when the bus recovery process completes.

Return

- `ESP_OK`: Bus recovery started
- `ESP_ERR_INVALID_STATE`: CAN driver is not in the bus-off state, or is not installed

esp_err_t **can_get_status_info** (`can_status_info_t *status_info`)

Get current status information of the CAN driver.

Return

- `ESP_OK`: Status information retrieved
- `ESP_ERR_INVALID_ARG`: Arguments are invalid
- `ESP_ERR_INVALID_STATE`: CAN driver is not installed

Parameters

- [out] `status_info`: Status information

esp_err_t **can_clear_transmit_queue** (`void`)

Clear the transmit queue.

This function will clear the transmit queue of all messages.

Note The transmit queue is automatically cleared when `can_stop()` or `can_initiate_recovery()` is called.

Return

- `ESP_OK`: Transmit queue cleared
- `ESP_ERR_INVALID_STATE`: CAN driver is not installed or TX queue is disabled

esp_err_t **can_clear_receive_queue** (void)

Clear the receive queue.

This function will clear the receive queue of all messages.

Note The receive queue is automatically cleared when `can_start()` is called.

Return

- `ESP_OK`: Transmit queue cleared
- `ESP_ERR_INVALID_STATE`: CAN driver is not installed

Structures

struct can_general_config_t

Structure for general configuration of the CAN driver.

Note Macro initializers are available for this structure

Public Members

can_mode_t **mode**

Mode of CAN controller

gpio_num_t **tx_io**

Transmit GPIO number

gpio_num_t **rx_io**

Receive GPIO number

gpio_num_t **clkout_io**

CLKOUT GPIO number (optional, set to -1 if unused)

gpio_num_t **bus_off_io**

Bus off indicator GPIO number (optional, set to -1 if unused)

uint32_t **tx_queue_len**

Number of messages TX queue can hold (set to 0 to disable TX Queue)

uint32_t **rx_queue_len**

Number of messages RX queue can hold

uint32_t **alerts_enabled**

Bit field of alerts to enable (see documentation)

uint32_t **clkout_divider**

CLKOUT divider. Can be 1 or any even number from 2 to 14 (optional, set to 0 if unused)

int **intr_flags**

Interrupt flags to set the priority of the driver's ISR. Note that to use the `ESP_INTR_FLAG_IRAM`, the `CONFIG_TWAI_ISR_IN_IRAM` option should be enabled first.

struct can_status_info_t

Structure to store status information of CAN driver.

Public Members

can_state_t **state**

Current state of CAN controller (Stopped/Running/Bus-Off/Recovery)

uint32_t **msgs_to_tx**

Number of messages queued for transmission or awaiting transmission completion

`uint32_t msgs_to_rx`
Number of messages in RX queue waiting to be read

`uint32_t tx_error_counter`
Current value of Transmit Error Counter

`uint32_t rx_error_counter`
Current value of Receive Error Counter

`uint32_t tx_failed_count`
Number of messages that failed transmissions

`uint32_t rx_missed_count`
Number of messages that were lost due to a full RX queue

`uint32_t arb_lost_count`
Number of instances arbitration was lost

`uint32_t bus_error_count`
Number of instances a bus error has occurred

Macros

CAN_IO_UNUSED
Marks GPIO as unused in CAN configuration

Enumerations

enum can_state_t
CAN driver states.

Values:

CAN_STATE_STOPPED
Stopped state. The CAN controller will not participate in any CAN bus activities

CAN_STATE_RUNNING
Running state. The CAN controller can transmit and receive messages

CAN_STATE_BUS_OFF
Bus-off state. The CAN controller cannot participate in bus activities until it has recovered

CAN_STATE_RECOVERING
Recovering state. The CAN controller is undergoing bus recovery

2.3.3 Digital To Analog Converter

Overview

ESP32 has two 8-bit DAC (digital to analog converter) channels, connected to GPIO25 (Channel 1) and GPIO26 (Channel 2).

The DAC driver allows these channels to be set to arbitrary voltages.

The DAC channels can also be driven with DMA-style written sample data, via the *I2S driver* when using the “built-in DAC mode” .

For other analog output options, see the *Sigma-delta Modulation module* and the *LED Control module*. Both these modules produce high frequency PWM output, which can be hardware low-pass filtered in order to generate a lower frequency analog output.

Application Example

Setting DAC channel 1 (GPIO 25) voltage to approx 0.78 of VDD_A voltage ($VDD * 200 / 255$). For VDD_A 3.3V, this is 2.59V:

```
#include <driver/dac.h>

...

dac_output_enable(DAC_CHANNEL_1);
dac_output_voltage(DAC_CHANNEL_1, 200);
```

API Reference

Header File

- [driver/include/driver/dac.h](#)

Functions

esp_err_t **dac_pad_get_io_num** (*dac_channel_t* channel, *gpio_num_t* **gpio_num*)

Get the gpio number of a specific DAC channel.

Return

- ESP_OK if success

Parameters

- channel: Channel to get the gpio number
- gpio_num: output buffer to hold the gpio number

esp_err_t **dac_output_voltage** (*dac_channel_t* channel, *uint8_t* *dac_value*)

Set DAC output voltage. DAC output is 8-bit. Maximum (255) corresponds to VDD3P3_RTC.

Note Need to configure DAC pad before calling this function. DAC channel 1 is attached to GPIO25, DAC channel 2 is attached to GPIO26

Return

- ESP_OK success

Parameters

- channel: DAC channel
- dac_value: DAC output value

esp_err_t **dac_output_enable** (*dac_channel_t* channel)

DAC pad output enable.

Note DAC channel 1 is attached to GPIO25, DAC channel 2 is attached to GPIO26 I2S left channel will be mapped to DAC channel 2 I2S right channel will be mapped to DAC channel 1

Parameters

- channel: DAC channel

esp_err_t **dac_output_disable** (*dac_channel_t* channel)

DAC pad output disable.

Note DAC channel 1 is attached to GPIO25, DAC channel 2 is attached to GPIO26

Return

- ESP_OK success

Parameters

- channel: DAC channel

esp_err_t **dac_i2s_enable** (void)

Enable DAC output data from I2S.

Return

- ESP_OK success

esp_err_t **dac_i2s_disable** (void)

Disable DAC output data from I2S.

Return

- ESP_OK success

esp_err_t **dac_cw_generator_enable** (void)

Enable cosine wave generator output.

Return

- ESP_OK success

esp_err_t **dac_cw_generator_disable** (void)

Disable cosine wave generator output.

Return

- ESP_OK success

esp_err_t **dac_cw_generator_config** (*dac_cw_config_t* *cw)

Config the cosine wave generator function in DAC module.

Return

- ESP_OK success

Parameters

- cw: Configuration.

GPIO Lookup Macros Some useful macros can be used to specified the GPIO number of a DAC channel, or vice versa. e.g.

1. DAC_CHANNEL_1_GPIO_NUM is the GPIO number of channel 1 (25);
2. DAC_GPIO26_CHANNEL is the channel number of GPIO 26 (channel 2).

Header File

- [soc/esp32/include/soc/dac_channel.h](#)

Macros

DAC_GPIO25_CHANNEL

DAC_CHANNEL_1_GPIO_NUM

DAC_GPIO26_CHANNEL

DAC_CHANNEL_2_GPIO_NUM

Header File

- [soc/include/hal/dac_types.h](#)

Structures

struct **dac_cw_config_t**

Config the cosine wave generator function in DAC module.

Public Members

dac_channel_t **en_ch**

Enable the cosine wave generator of DAC channel.

dac_cw_scale_t **scale**

Set the amplitude of the cosine wave generator output.

***dac_cw_phase_t* phase**

Set the phase of the cosine wave generator output.

uint32_t freq

Set frequency of cosine wave generator output. Range: 130(130Hz) ~ 55000(100KHz).

int8_t offset

Set the voltage value of the DC component of the cosine wave generator output. Note: Unreasonable settings can cause waveform to be oversaturated. Range: -128 ~ 127.

Enumerations**enum dac_channel_t**

Values:

DAC_CHANNEL_1 = 0

DAC channel 1 is GPIO25(ESP32) / GPIO17(ESP32S2BETA)

DAC_CHANNEL_2 = 1

DAC channel 2 is GPIO26(ESP32) / GPIO18(ESP32S2BETA)

DAC_CHANNEL_MAX

enum dac_cw_scale_t

The multiple of the amplitude of the cosine wave generator. The max amplitude is VDD3P3_RTC.

Values:

DAC_CW_SCALE_1 = 0x0

1/1. Default.

DAC_CW_SCALE_2 = 0x1

1/2.

DAC_CW_SCALE_4 = 0x2

1/4.

DAC_CW_SCALE_8 = 0x3

1/8.

enum dac_cw_phase_t

Set the phase of the cosine wave generator output.

Values:

DAC_CW_PHASE_0 = 0x2

Phase shift +0°

DAC_CW_PHASE_180 = 0x3

Phase shift +180°

2.3.4 GPIO & RTC GPIO

Overview

The ESP32 chip features 40 physical GPIO pads. Some GPIO pads cannot be used or do not have the corresponding pin on the chip package(refer to technical reference manual). Each pad can be used as a general purpose I/O or can be connected to an internal peripheral signal.

- Note that GPIO6-11 are usually used for SPI flash.
- GPIO34-39 can only be set as input mode and do not have software pullup or pulldown functions.

There is also separate “RTC GPIO” support, which functions when GPIOs are routed to the “RTC” low-power and analog subsystem. These pin functions can be used when in deep sleep, when the *Ultra Low Power co-processor* is running, or when analog functions such as ADC/DAC/etc are in use.

Application Example

GPIO output and input interrupt example: [peripherals/gpio](#).

API Reference - Normal GPIO

Header File

- [driver/include/driver/gpio.h](#)

Functions

esp_err_t **gpio_config** (*const gpio_config_t* *pGPIOConfig)
GPIO common configuration.

Configure GPIO's Mode, pull-up, PullDown, IntrType

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- pGPIOConfig: Pointer to GPIO configure struct

esp_err_t **gpio_reset_pin** (*gpio_num_t* gpio_num)

Reset an gpio to default state (select gpio function, enable pullup and disable input and output).

Note This function also configures the IOMUX for this pin to the GPIO function, and disconnects any other peripheral output configured via GPIO Matrix.

Return Always return ESP_OK.

Parameters

- gpio_num: GPIO number.

esp_err_t **gpio_set_intr_type** (*gpio_num_t* gpio_num, *gpio_int_type_t* intr_type)

GPIO set interrupt trigger type.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- gpio_num: GPIO number. If you want to set the trigger type of e.g. of GPIO16, gpio_num should be GPIO_NUM_16 (16);
- intr_type: Interrupt type, select from gpio_int_type_t

esp_err_t **gpio_intr_enable** (*gpio_num_t* gpio_num)

Enable GPIO module interrupt signal.

Note Please do not use the interrupt of GPIO36 and GPIO39 when using ADC or Wi-Fi with sleep mode enabled. Please refer to the comments of `adc1_get_raw`. Please refer to section 3.11 of 'ECO_and_Workarounds_for_Bugs_in_ESP32' for the description of this issue. As a workaround, call `adc_power_acquire()` in the app. This will result in higher power consumption (by ~1mA), but will remove the glitches on GPIO36 and GPIO39.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- gpio_num: GPIO number. If you want to enable an interrupt on e.g. GPIO16, gpio_num should be GPIO_NUM_16 (16);

esp_err_t **gpio_intr_disable** (*gpio_num_t* gpio_num)

Disable GPIO module interrupt signal.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `gpio_num`: GPIO number. If you want to disable the interrupt of e.g. GPIO16, `gpio_num` should be `GPIO_NUM_16` (16);

esp_err_t **gpio_set_level** (*gpio_num_t* `gpio_num`, *uint32_t* `level`)

GPIO set output level.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` GPIO number error

Parameters

- `gpio_num`: GPIO number. If you want to set the output level of e.g. GPIO16, `gpio_num` should be `GPIO_NUM_16` (16);
- `level`: Output level. 0: low ; 1: high

int **gpio_get_level** (*gpio_num_t* `gpio_num`)

GPIO get input level.

Warning If the pad is not configured for input (or input and output) the returned value is always 0.

Return

- 0 the GPIO input level is 0
- 1 the GPIO input level is 1

Parameters

- `gpio_num`: GPIO number. If you want to get the logic level of e.g. pin GPIO16, `gpio_num` should be `GPIO_NUM_16` (16);

esp_err_t **gpio_set_direction** (*gpio_num_t* `gpio_num`, *gpio_mode_t* `mode`)

GPIO set direction.

Configure GPIO direction, such as `output_only`, `input_only`, `output_and_input`

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` GPIO error

Parameters

- `gpio_num`: Configure GPIO pins number, it should be GPIO number. If you want to set direction of e.g. GPIO16, `gpio_num` should be `GPIO_NUM_16` (16);
- `mode`: GPIO direction

esp_err_t **gpio_set_pull_mode** (*gpio_num_t* `gpio_num`, *gpio_pull_mode_t* `pull`)

Configure GPIO pull-up/pull-down resistors.

Only pins that support both input & output have integrated pull-up and pull-down resistors. Input-only GPIOs 34-39 do not.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` : Parameter error

Parameters

- `gpio_num`: GPIO number. If you want to set pull up or down mode for e.g. GPIO16, `gpio_num` should be `GPIO_NUM_16` (16);
- `pull`: GPIO pull up/down mode.

esp_err_t **gpio_wakeup_enable** (*gpio_num_t* `gpio_num`, *gpio_int_type_t* `intr_type`)

Enable GPIO wake-up function.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `gpio_num`: GPIO number.
- `intr_type`: GPIO wake-up type. Only `GPIO_INTR_LOW_LEVEL` or `GPIO_INTR_HIGH_LEVEL` can be used.

esp_err_t **gpio_wakeup_disable** (*gpio_num_t* *gpio_num*)

Disable GPIO wake-up function.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- *gpio_num*: GPIO number

esp_err_t **gpio_isr_register** (void (**fn*)) void *

, void **arg*, int *intr_alloc_flags*, *gpio_isr_handle_t* **handle* Register GPIO interrupt handler, the handler is an ISR. The handler will be attached to the same CPU core that this function is running on.

This ISR function is called whenever any GPIO interrupt occurs. See the alternative `gpio_install_isr_service()` and `gpio_isr_handler_add()` API in order to have the driver support per-GPIO ISRs.

To disable or remove the ISR, pass the returned handle to the *interrupt allocation functions*.

Parameters

- *fn*: Interrupt handler function.
- *intr_alloc_flags*: Flags used to allocate the interrupt. One or multiple (ORred) ESP_INTR_FLAG_* values. See `esp_intr_alloc.h` for more info.
- *arg*: Parameter for handler function
- *handle*: Pointer to return handle. If non-NULL, a handle for the interrupt will be returned here.

Return

- ESP_OK Success ;
- ESP_ERR_INVALID_ARG GPIO error
- ESP_ERR_NOT_FOUND No free interrupt found with the specified flags

esp_err_t **gpio_pullup_en** (*gpio_num_t* *gpio_num*)

Enable pull-up on GPIO.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- *gpio_num*: GPIO number

esp_err_t **gpio_pullup_dis** (*gpio_num_t* *gpio_num*)

Disable pull-up on GPIO.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- *gpio_num*: GPIO number

esp_err_t **gpiopulldown_en** (*gpio_num_t* *gpio_num*)

Enable pull-down on GPIO.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- *gpio_num*: GPIO number

esp_err_t **gpiopulldown_dis** (*gpio_num_t* *gpio_num*)

Disable pull-down on GPIO.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- *gpio_num*: GPIO number

esp_err_t **gpio_install_isr_service** (int *intr_alloc_flags*)

Install the driver's GPIO ISR handler service, which allows per-pin GPIO interrupt handlers.

This function is incompatible with `gpio_isr_register()` - if that function is used, a single global ISR is registered for all GPIO interrupts. If this function is used, the ISR service provides a global GPIO ISR and individual pin handlers are registered via the `gpio_isr_handler_add()` function.

Return

- ESP_OK Success
- ESP_ERR_NO_MEM No memory to install this service
- ESP_ERR_INVALID_STATE ISR service already installed.
- ESP_ERR_NOT_FOUND No free interrupt found with the specified flags
- ESP_ERR_INVALID_ARG GPIO error

Parameters

- *intr_alloc_flags*: Flags used to allocate the interrupt. One or multiple (ORred) ESP_INTR_FLAG_* values. See `esp_intr_alloc.h` for more info.

void **gpio_uninstall_isr_service** (void)

Uninstall the driver's GPIO ISR service, freeing related resources.

esp_err_t **gpio_isr_handler_add** (*gpio_num_t* *gpio_num*, *gpio_isr_t* *isr_handler*, void **args*)

Add ISR handler for the corresponding GPIO pin.

Call this function after using `gpio_install_isr_service()` to install the driver's GPIO ISR handler service.

The pin ISR handlers no longer need to be declared with `IRAM_ATTR`, unless you pass the `ESP_INTR_FLAG_IRAM` flag when allocating the ISR in `gpio_install_isr_service()`.

This ISR handler will be called from an ISR. So there is a stack size limit (configurable as "ISR stack size" in `menuconfig`). This limit is smaller compared to a global GPIO interrupt handler due to the additional level of indirection.

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE Wrong state, the ISR service has not been initialized.
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- *gpio_num*: GPIO number
- *isr_handler*: ISR handler function for the corresponding GPIO number.
- *args*: parameter for ISR handler.

esp_err_t **gpio_isr_handler_remove** (*gpio_num_t* *gpio_num*)

Remove ISR handler for the corresponding GPIO pin.

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE Wrong state, the ISR service has not been initialized.
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- *gpio_num*: GPIO number

esp_err_t **gpio_set_drive_capability** (*gpio_num_t* *gpio_num*, *gpio_drive_cap_t* *strength*)

Set GPIO pad drive capability.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- *gpio_num*: GPIO number, only support output GPIOs
- *strength*: Drive capability of the pad

esp_err_t **gpio_get_drive_capability** (*gpio_num_t* *gpio_num*, *gpio_drive_cap_t* **strength*)

Get GPIO pad drive capability.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `gpio_num`: GPIO number, only support output GPIOs
- `strength`: Pointer to accept drive capability of the pad

esp_err_t **gpio_hold_en** (*gpio_num_t* *gpio_num*)

Enable gpio pad hold function.

The gpio pad hold function works in both input and output modes, but must be output-capable gpios. If pad hold enabled: in output mode: the output level of the pad will be force locked and can not be changed. in input mode: the input value read will not change, regardless the changes of input signal.

The state of digital gpio cannot be held during Deep-sleep, and it will resume the hold function when the chip wakes up from Deep-sleep. If the digital gpio also needs to be held during Deep-sleep, `gpio_deep_sleep_hold_en` should also be called.

Power down or call `gpio_hold_dis` will disable this function.

Return

- ESP_OK Success
- ESP_ERR_NOT_SUPPORTED Not support pad hold function

Parameters

- `gpio_num`: GPIO number, only support output-capable GPIOs

esp_err_t **gpio_hold_dis** (*gpio_num_t* *gpio_num*)

Disable gpio pad hold function.

When the chip is woken up from Deep-sleep, the gpio will be set to the default mode, so, the gpio will output the default level if this function is called. If you don't want the level changes, the gpio should be configured to a known state before this function is called. e.g. If you hold gpio18 high during Deep-sleep, after the chip is woken up and `gpio_hold_dis` is called, gpio18 will output low level(because gpio18 is input mode by default). If you don't want this behavior, you should configure gpio18 as output mode and set it to high level before calling `gpio_hold_dis`.

Return

- ESP_OK Success
- ESP_ERR_NOT_SUPPORTED Not support pad hold function

Parameters

- `gpio_num`: GPIO number, only support output-capable GPIOs

void **gpio_deep_sleep_hold_en** (void)

Enable all digital gpio pad hold function during Deep-sleep.

When the chip is in Deep-sleep mode, all digital gpio will hold the state before sleep, and when the chip is woken up, the status of digital gpio will not be held. Note that the pad hold feature only works when the chip is in Deep-sleep mode, when not in sleep mode, the digital gpio state can be changed even you have called this function.

Power down or call `gpio_hold_dis` will disable this function, otherwise, the digital gpio hold feature works as long as the chip enter Deep-sleep.

void **gpio_deep_sleep_hold_dis** (void)

Disable all digital gpio pad hold function during Deep-sleep.

void **gpio_iomux_in** (uint32_t *gpio_num*, uint32_t *signal_idx*)

Set pad input to a peripheral signal through the IOMUX.

Parameters

- `gpio_num`: GPIO number of the pad.
- `signal_idx`: Peripheral signal id to input. One of the `*_IN_IDX` signals in `soc/gpio_sig_map.h`.

void **gpio_iomux_out** (uint8_t *gpio_num*, int *func*, bool *oen_inv*)

Set peripheral output to an GPIO pad through the IOMUX.

Parameters

- `gpio_num`: `gpio_num` GPIO number of the pad.
- `func`: The function number of the peripheral pin to output pin. One of the `FUNC_X_*` of specified pin (X) in `soc/io_mux_reg.h`.
- `oen_inv`: True if the output enable needs to be inverted, otherwise False.

Type Definitions

```
typedef intr_handle_t gpio_isr_handle_t
```

Header File

- `soc/include/hal/gpio_types.h`

Structures

```
struct gpio_config_t
```

Configuration parameters of GPIO pad for `gpio_config` function.

Public Members

```
uint64_t pin_bit_mask
```

GPIO pin: set with bit mask, each bit maps to a GPIO

```
gpio_mode_t mode
```

GPIO mode: set input/output mode

```
gpio_pullup_t pull_up_en
```

GPIO pull-up

```
gpio_pulldown_t pull_down_en
```

GPIO pull-down

```
gpio_int_type_t intr_type
```

GPIO interrupt type

Type Definitions

```
typedef void (*gpio_isr_t)(void *)
```

Enumerations

```
enum gpio_port_t
```

Values:

```
GPIO_PORT_0 = 0
```

```
GPIO_PORT_MAX
```

```
enum gpio_num_t
```

Values:

```
GPIO_NUM_NC = -1
```

Use to signal not connected to S/W

```
GPIO_NUM_0 = 0
```

GPIO0, input and output

```
GPIO_NUM_1 = 1
```

GPIO1, input and output

```
GPIO_NUM_2 = 2
```

GPIO2, input and output

GPIO_NUM_3 = 3
GPIO3, input and output

GPIO_NUM_4 = 4
GPIO4, input and output

GPIO_NUM_5 = 5
GPIO5, input and output

GPIO_NUM_6 = 6
GPIO6, input and output

GPIO_NUM_7 = 7
GPIO7, input and output

GPIO_NUM_8 = 8
GPIO8, input and output

GPIO_NUM_9 = 9
GPIO9, input and output

GPIO_NUM_10 = 10
GPIO10, input and output

GPIO_NUM_11 = 11
GPIO11, input and output

GPIO_NUM_12 = 12
GPIO12, input and output

GPIO_NUM_13 = 13
GPIO13, input and output

GPIO_NUM_14 = 14
GPIO14, input and output

GPIO_NUM_15 = 15
GPIO15, input and output

GPIO_NUM_16 = 16
GPIO16, input and output

GPIO_NUM_17 = 17
GPIO17, input and output

GPIO_NUM_18 = 18
GPIO18, input and output

GPIO_NUM_19 = 19
GPIO19, input and output

GPIO_NUM_20 = 20
GPIO20, input and output

GPIO_NUM_21 = 21
GPIO21, input and output

GPIO_NUM_26 = 26
GPIO26, input and output

GPIO_NUM_27 = 27
GPIO27, input and output

GPIO_NUM_28 = 28
GPIO28, input and output

GPIO_NUM_29 = 29
GPIO29, input and output

GPIO_NUM_30 = 30
GPIO30, input and output

GPIO_NUM_31 = 31
GPIO31, input and output

GPIO_NUM_32 = 32
GPIO32, input and output

GPIO_NUM_33 = 33
GPIO33, input and output

GPIO_NUM_34 = 34
GPIO34, input mode only(ESP32) / input and output(ESP32-S2)

GPIO_NUM_35 = 35
GPIO35, input mode only(ESP32) / input and output(ESP32-S2)

GPIO_NUM_36 = 36
GPIO36, input mode only(ESP32) / input and output(ESP32-S2)

GPIO_NUM_37 = 37
GPIO37, input mode only(ESP32) / input and output(ESP32-S2)

GPIO_NUM_38 = 38
GPIO38, input mode only(ESP32) / input and output(ESP32-S2)

GPIO_NUM_39 = 39
GPIO39, input mode only(ESP32) / input and output(ESP32-S2)

GPIO_NUM_MAX

enum gpio_int_type_t

Values:

GPIO_INTR_DISABLE = 0
Disable GPIO interrupt

GPIO_INTR_POSEDGE = 1
GPIO interrupt type : rising edge

GPIO_INTR_NEGEDGE = 2
GPIO interrupt type : falling edge

GPIO_INTR_ANYEDGE = 3
GPIO interrupt type : both rising and falling edge

GPIO_INTR_LOW_LEVEL = 4
GPIO interrupt type : input low level trigger

GPIO_INTR_HIGH_LEVEL = 5
GPIO interrupt type : input high level trigger

GPIO_INTR_MAX

enum gpio_mode_t

Values:

GPIO_MODE_DISABLE = GPIO_MODE_DEF_DISABLE
GPIO mode : disable input and output

GPIO_MODE_INPUT = GPIO_MODE_DEF_INPUT
GPIO mode : input only

GPIO_MODE_OUTPUT = GPIO_MODE_DEF_OUTPUT
GPIO mode : output only mode

GPIO_MODE_OUTPUT_OD = ((GPIO_MODE_DEF_OUTPUT) | (GPIO_MODE_DEF_OD))
GPIO mode : output only with open-drain mode

GPIO_MODE_INPUT_OUTPUT_OD = ((GPIO_MODE_DEF_INPUT) | (GPIO_MODE_DEF_OUTPUT) | (GPIO_MODE_...
GPIO mode : output and input with open-drain mode

GPIO_MODE_INPUT_OUTPUT = ((GPIO_MODE_DEF_INPUT) | (GPIO_MODE_DEF_OUTPUT))
GPIO mode : output and input mode

enum gpio_pullup_t

Values:

GPIO_PULLUP_DISABLE = 0x0
Disable GPIO pull-up resistor

GPIO_PULLUP_ENABLE = 0x1
Enable GPIO pull-up resistor

enum gpiopulldown_t

Values:

GPIO_PULLDOWN_DISABLE = 0x0
Disable GPIO pull-down resistor

GPIO_PULLDOWN_ENABLE = 0x1
Enable GPIO pull-down resistor

enum gpio_pull_mode_t

Values:

GPIO_PULLUP_ONLY
Pad pull up

GPIO_PULLDOWN_ONLY
Pad pull down

GPIO_PULLUP_PULLDOWN
Pad pull up + pull down

GPIO_FLOATING
Pad floating

enum gpio_drive_cap_t

Values:

GPIO_DRIVE_CAP_0 = 0
Pad drive capability: weak

GPIO_DRIVE_CAP_1 = 1
Pad drive capability: stronger

GPIO_DRIVE_CAP_2 = 2
Pad drive capability: medium

GPIO_DRIVE_CAP_DEFAULT = 2
Pad drive capability: medium

GPIO_DRIVE_CAP_3 = 3
Pad drive capability: strongest

GPIO_DRIVE_CAP_MAX

API Reference - RTC GPIO

Header File

- [driver/include/driver/rtc_io.h](#)

Functions

static bool `rtc_gpio_is_valid_gpio` (*gpio_num_t* gpio_num)

Determine if the specified GPIO is a valid RTC GPIO.

Return true if GPIO is valid for RTC GPIO use. false otherwise.

Parameters

- gpio_num: GPIO number

static int `rtc_io_number_get` (*gpio_num_t* gpio_num)

Get RTC IO index number by gpio number.

Return >=0: Index of rtcio. -1 : The gpio is not rtcio.

Parameters

- gpio_num: GPIO number

esp_err_t `rtc_gpio_init` (*gpio_num_t* gpio_num)

Init a GPIO as RTC GPIO.

This function must be called when initializing a pad for an analog function.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

Parameters

- gpio_num: GPIO number (e.g. GPIO_NUM_12)

esp_err_t `rtc_gpio_deinit` (*gpio_num_t* gpio_num)

Init a GPIO as digital GPIO.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

Parameters

- gpio_num: GPIO number (e.g. GPIO_NUM_12)

uint32_t `rtc_gpio_get_level` (*gpio_num_t* gpio_num)

Get the RTC IO input level.

Return

- 1 High level
- 0 Low level
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

Parameters

- gpio_num: GPIO number (e.g. GPIO_NUM_12)

esp_err_t `rtc_gpio_set_level` (*gpio_num_t* gpio_num, *uint32_t* level)

Set the RTC IO output level.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

Parameters

- gpio_num: GPIO number (e.g. GPIO_NUM_12)
- level: output level

esp_err_t `rtc_gpio_set_direction` (*gpio_num_t* gpio_num, *rtc_gpio_mode_t* mode)

RTC GPIO set direction.

Configure RTC GPIO direction, such as output only, input only, output and input.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

Parameters

- gpio_num: GPIO number (e.g. GPIO_NUM_12)
- mode: GPIO direction

esp_err_t rtc_gpio_set_direction_in_sleep (*gpio_num_t* gpio_num, *rtc_gpio_mode_t* mode)

RTC GPIO set direction in deep sleep mode or disable sleep status (default). In some application scenarios, IO needs to have another states during deep sleep.

NOTE: ESP32 support INPUT_ONLY mode. ESP32S2 support INPUT_ONLY, OUTPUT_ONLY, INPUT_OUTPUT mode.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

Parameters

- gpio_num: GPIO number (e.g. GPIO_NUM_12)
- mode: GPIO direction

esp_err_t rtc_gpio_pullup_en (*gpio_num_t* gpio_num)

RTC GPIO pullup enable.

This function only works for RTC IOs. In general, call gpio_pullup_en, which will work both for normal GPIOs and RTC IOs.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

Parameters

- gpio_num: GPIO number (e.g. GPIO_NUM_12)

esp_err_t rtc_gpio_pulldown_en (*gpio_num_t* gpio_num)

RTC GPIO pulldown enable.

This function only works for RTC IOs. In general, call gpio_pulldown_en, which will work both for normal GPIOs and RTC IOs.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

Parameters

- gpio_num: GPIO number (e.g. GPIO_NUM_12)

esp_err_t rtc_gpio_pullup_dis (*gpio_num_t* gpio_num)

RTC GPIO pullup disable.

This function only works for RTC IOs. In general, call gpio_pullup_dis, which will work both for normal GPIOs and RTC IOs.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

Parameters

- gpio_num: GPIO number (e.g. GPIO_NUM_12)

esp_err_t rtc_gpio_pulldown_dis (*gpio_num_t* gpio_num)

RTC GPIO pulldown disable.

This function only works for RTC IOs. In general, call gpio_pulldown_dis, which will work both for normal GPIOs and RTC IOs.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

Parameters

- gpio_num: GPIO number (e.g. GPIO_NUM_12)

esp_err_t rtc_gpio_hold_en (*gpio_num_t* gpio_num)

Enable hold function on an RTC IO pad.

Enabling HOLD function will cause the pad to latch current values of input enable, output enable, output value, function, drive strength values. This function is useful when going into light or deep sleep mode to prevent the pin configuration from changing.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

Parameters

- `gpio_num`: GPIO number (e.g. GPIO_NUM_12)

esp_err_t rtc_gpio_hold_dis (gpio_num_t gpio_num)

Disable hold function on an RTC IO pad.

Disabling hold function will allow the pad receive the values of input enable, output enable, output value, function, drive strength from RTC_IO peripheral.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

Parameters

- `gpio_num`: GPIO number (e.g. GPIO_NUM_12)

esp_err_t rtc_gpio_isolate (gpio_num_t gpio_num)

Helper function to disconnect internal circuits from an RTC IO This function disables input, output, pullup, pulldown, and enables hold feature for an RTC IO. Use this function if an RTC IO needs to be disconnected from internal circuits in deep sleep, to minimize leakage current.

In particular, for ESP32-WROVER module, call `rtc_gpio_isolate(GPIO_NUM_12)` before entering deep sleep, to reduce deep sleep current.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if GPIO is not an RTC IO

Parameters

- `gpio_num`: GPIO number (e.g. GPIO_NUM_12).

esp_err_t rtc_gpio_force_hold_all (void)

Enable force hold signal for all RTC IOs.

Each RTC pad has a “force hold” input signal from the RTC controller. If this signal is set, pad latches current values of input enable, function, output enable, and other signals which come from the RTC mux. Force hold signal is enabled before going into deep sleep for pins which are used for EXT1 wakeup.

esp_err_t rtc_gpio_force_hold_dis_all (void)

Disable force hold signal for all RTC IOs.

esp_err_t rtc_gpio_set_drive_capability (gpio_num_t gpio_num, gpio_drive_cap_t strength)

Set RTC GPIO pad drive capability.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `gpio_num`: GPIO number, only support output GPIOs
- `strength`: Drive capability of the pad

*esp_err_t rtc_gpio_get_drive_capability (gpio_num_t gpio_num, gpio_drive_cap_t *strength)*

Get RTC GPIO pad drive capability.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `gpio_num`: GPIO number, only support output GPIOs
- `strength`: Pointer to accept drive capability of the pad

esp_err_t **rtc_gpio_wakeup_enable** (*gpio_num_t* gpio_num, *gpio_int_type_t* intr_type)

Enable wakeup from sleep mode using specific GPIO.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if gpio_num is not an RTC IO, or intr_type is not one of GPIO_INTR_HIGH_LEVEL, GPIO_INTR_LOW_LEVEL.

Parameters

- gpio_num: GPIO number
- intr_type: Wakeup on high level (GPIO_INTR_HIGH_LEVEL) or low level (GPIO_INTR_LOW_LEVEL)

esp_err_t **rtc_gpio_wakeup_disable** (*gpio_num_t* gpio_num)

Disable wakeup from sleep mode using specific GPIO.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if gpio_num is not an RTC IO

Parameters

- gpio_num: GPIO number

Macros

RTC_GPIO_IS_VALID_GPIO (gpio_num)

Header File

- [soc/include/hal/rtc_io_types.h](#)

Enumerations

enum rtc_gpio_mode_t

RTCIO output/input mode type.

Values:

RTC_GPIO_MODE_INPUT_ONLY

Pad input

RTC_GPIO_MODE_OUTPUT_ONLY

Pad output

RTC_GPIO_MODE_INPUT_OUTPUT

Pad input + output

RTC_GPIO_MODE_DISABLED

Pad (output + input) disable

RTC_GPIO_MODE_OUTPUT_OD

Pad open-drain output

RTC_GPIO_MODE_INPUT_OUTPUT_OD

Pad input + open-drain output

2.3.5 I2C Driver

Overview

I2C is a serial, synchronous, half-duplex communication protocol that allows co-existence of multiple masters and slaves on the same bus. The I2C bus consists of two lines: serial data line (SDA) and serial clock (SCL). Both lines require pull-up resistors.

With such advantages as simplicity and low manufacturing cost, I2C is mostly used for communication of low-speed peripheral devices over short distances (within one foot).

ESP32 has two I2C controllers (also referred to as ports) which are responsible for handling communications on two I2C buses. Each I2C controller can operate as master or slave. As an example, one controller can act as a master and the other as a slave at the same time.

Driver Features

I2C driver governs communications of devices over the I2C bus. The driver supports the following features:

- Reading and writing bytes in Master mode
- Slave mode
- Reading and writing to registers which are in turn read/written by the master

Driver Usage

The following sections describe typical steps of configuring and operating the I2C driver:

1. *Configuration* - set the initialization parameters (master or slave mode, GPIO pins for SDA and SCL, clock speed, etc.)
2. *Install Driver*- activate the driver on one of the two I2C controllers as a master or slave
3. Depending on whether you configure the driver for a master or slave, choose the appropriate item
 - a) *Communication as Master* - handle communications (master)
 - b) *Communication as Slave* - respond to messages from the master (slave)
4. *Interrupt Handling* - configure and service I2C interrupts
5. *Customized Configuration* - adjust default I2C communication parameters (timings, bit order, etc.)
6. *Error Handling* - how to recognize and handle driver configuration and communication errors
7. *Delete Driver*- release resources used by the I2C driver when communication ends

Configuration To establish I2C communication, start by configuring the driver. This is done by setting the parameters of the structure `i2c_config_t`:

- Set I2C **mode of operation** - slave or master from `i2c_mode_t`
- Configure **communication pins**
 - Assign GPIO pins for SDA and SCL signals
 - Set whether to enable ESP32' s internal pull-ups
- (Master only) Set I2C **clock speed**
- (Slave only) Configure the following
 - Whether to enable **10 bit address mode**
 - Define **slave address**

After that, initialize the configuration for a given I2C port. For this, call the function `i2c_param_config()` and pass to it the port number and the structure `i2c_config_t`.

At this stage, `i2c_param_config()` also sets a few other I2C configuration parameters to default values that are defined by the I2C specification. For more details on the values and how to modify them, see *Customized Configuration*.

Install Driver After the I2C driver is configured, install it by calling the function `i2c_driver_install()` with the following parameters:

- Port number, one of the two port numbers from `i2c_port_t`
- Master or slave, selected from `i2c_mode_t`
- (Slave only) Size of buffers to allocate for sending and receiving data. As I2C is a master-centric bus, data can only go from the slave to the master at the master' s request. Therefore, the slave will usually have a send buffer where the slave application writes data. The data remains in the send buffer to be read by the master at the master' s own discretion.
- Flags for allocating the interrupt (see `ESP_INTR_FLAG_*` values in `esp32/include/esp_intr_alloc.h`)

Communication as Master After installing the I2C driver, ESP32 is ready to communicate with other I2C devices.

ESP32's I2C controller operating as master is responsible for establishing communication with I2C slave devices and sending commands to trigger a slave to action, for example, to take a measurement and send the readings back to the master.

For better process organization, the driver provides a container, called a “command link”, that should be populated with a sequence of commands and then passed to the I2C controller for execution.

Master Write The example below shows how to build a command link for an I2C master to send n bytes to a slave.

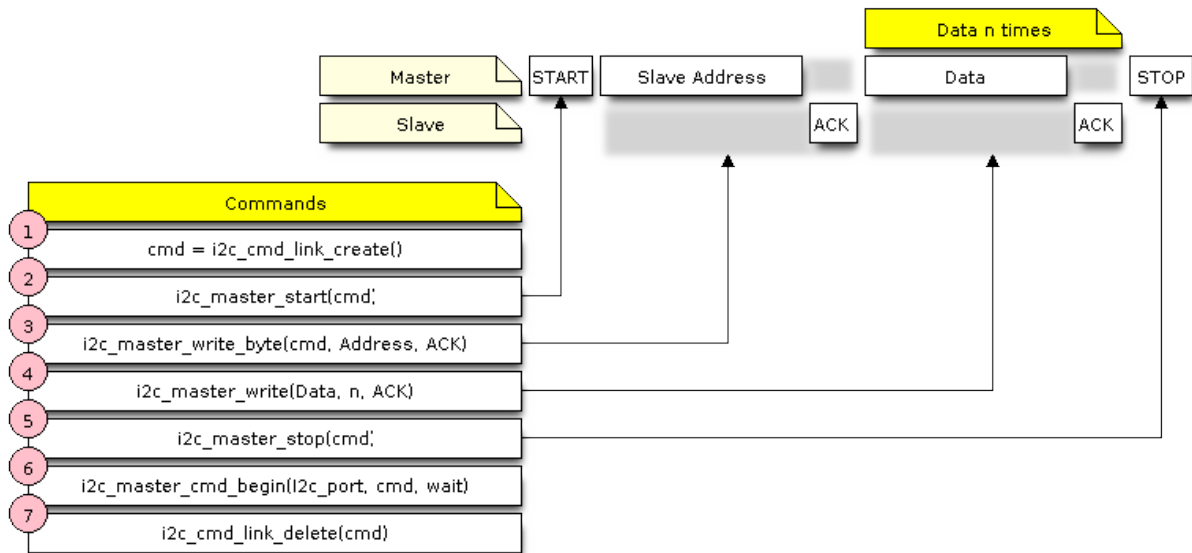


Fig. 12: I2C command link - master write example

The following describes how a command link for a “master write” is set up and what comes inside:

1. Create a command link with `i2c_cmd_link_create()`.
Then, populate it with the series of data to be sent to the slave:
 - a) **Start bit** - `i2c_master_start()`
 - b) **Slave address** - `i2c_master_write_byte()`. The single byte address is provided as an argument of this function call.
 - c) **Data** - One or more bytes as an argument of `i2c_master_write()`
 - d) **Stop bit** - `i2c_master_stop()`
 Both functions `i2c_master_write_byte()` and `i2c_master_write()` have an additional argument specifying whether the master should ensure that it has received the ACK bit.
2. Trigger the execution of the command link by I2C controller by calling `i2c_master_cmd_begin()`. Once the execution is triggered, the command link cannot be modified.
3. After the commands are transmitted, release the resources used by the command link by calling `i2c_cmd_link_delete()`.

Master Read The example below shows how to build a command link for an I2C master to read n bytes from a slave.

Compared to writing data, the command link is populated in Step 4 not with `i2c_master_write...` functions but with `i2c_master_read_byte()` and / or `i2c_master_read()`. Also, the last read in Step 5 is configured so that the master does not provide the ACK bit.

Indicating Write or Read After sending a slave address (see Step 3 on both diagrams above), the master either writes or reads from the slave.

The information on what the master will actually do is hidden in the least significant bit of the slave's address.

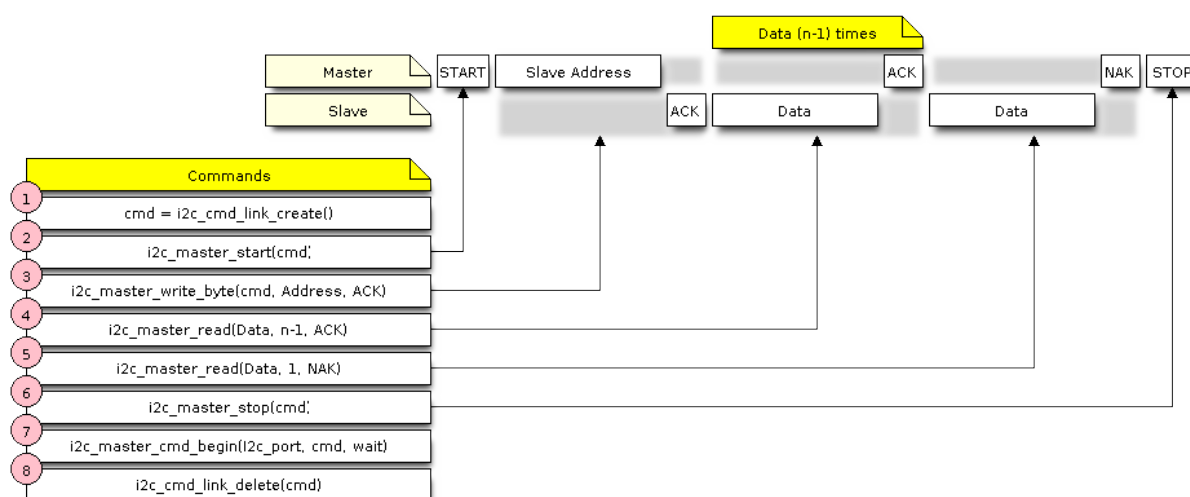


Fig. 13: I2C command link - master read example

For this reason, the command link sent by the master to write data to the slave contains the address (`ESP_SLAVE_ADDR << 1`) | `I2C_MASTER_WRITE` and looks as follows:

```
i2c_master_write_byte(cmd, (ESP_SLAVE_ADDR << 1) | I2C_MASTER_WRITE, ACK_EN);
```

Likewise, the command link to read from the slave looks as follows:

```
i2c_master_write_byte(cmd, (ESP_SLAVE_ADDR << 1) | I2C_MASTER_READ, ACK_EN);
```

Communication as Slave After installing the I2C driver, ESP32 is ready to communicate with other I2C devices.

The API provides the following functions for slaves

- `i2c_slave_read_buffer()`
Whenever the master writes data to the slave, the slave will automatically store it in the receive buffer. This allows the slave application to call the function `i2c_slave_read_buffer()` at its own discretion. This function also has a parameter to specify block time if no data is in the receive buffer. This will allow the slave application to wait with a specified timeout for data to arrive to the buffer.
- `i2c_slave_write_buffer()`
The send buffer is used to store all the data that the slave wants to send to the master in FIFO order. The data stays there until the master requests for it. The function `i2c_slave_write_buffer()` has a parameter to specify block time if the send buffer is full. This will allow the slave application to wait with a specified timeout for the adequate amount of space to become available in the send buffer.

A code example showing how to use these functions can be found in [peripherals/i2c](#).

Interrupt Handling During driver installation, an interrupt handler is installed by default. However, you can register your own interrupt handler instead of the default one by calling the function `i2c_isr_register()`. When implementing your own interrupt handler, refer to the [ESP32 Technical Reference Manual](#) for the description of interrupts triggered by the I2C controller.

To delete an interrupt handler, call `i2c_isr_free()`.

Customized Configuration As mentioned at the end of Section [Configuration](#), when the function `i2c_param_config()` initializes the driver configuration for an I2C port, it also sets several I2C communication parameters to default values defined in the [I2C specification](#). Some other related parameters are pre-configured in registers of the I2C controller.

All these parameters can be changed to user-defined values by calling dedicated functions given in the table below. Please note that the timing values are defined in APB clock cycles. The frequency of APB is specified in `I2C_APB_CLK_FREQ`.

Table 1: Other Configurable I2C Communication Parameters

| Parameters to Change | Function |
|--|-------------------------------------|
| High time and low time for SCL pulses | <code>i2c_set_period()</code> |
| SCL and SDA signal timing used during generation of start signals | <code>i2c_set_start_timing()</code> |
| SCL and SDA signal timing used during generation of stop signals | <code>i2c_set_stop_timing()</code> |
| Timing relationship between SCL and SDA signals when slave samples, as well as when master toggles | <code>i2c_set_data_timing()</code> |
| I2C timeout | <code>i2c_set_timeout()</code> |
| Choice between transmitting / receiving the LSB or MSB first, choose one of the modes defined in <code>i2c_trans_mode_t</code> | <code>i2c_set_data_mode()</code> |

Each of the above functions has a `_get_` counterpart to check the currently set value. For example, to check the I2C timeout value, call `i2c_get_timeout()`.

To check the default parameter values which are set during the driver configuration process, please refer to the file `driver/i2c.c` and look for defines with the suffix `_DEFAULT`.

You can also select different pins for SDA and SCL signals and alter the configuration of pull-ups with the function `i2c_set_pin()`. If you want to modify already entered values, use the function `i2c_param_config()`.

Note: ESP32's internal pull-ups are in the range of tens of kOhm, which is, in most cases, insufficient for use as I2C pull-ups. Users are advised to use external pull-ups with values described in the [I2C specification](#).

Error Handling The majority of I2C driver functions either return `ESP_OK` on successful completion or a specific error code on failure. It is a good practice to always check the returned values and implement error handling. The driver also prints out log messages that contain error details, e.g., when checking the validity of entered configuration. For details please refer to the file `driver/i2c.c` and look for defines with the suffix `_ERR_STR`.

Use dedicated interrupts to capture communication failures. For instance, if a slave stretches the clock for too long while preparing the data to send back to master, the interrupt `I2C_TIME_OUT_INT` will be triggered. For detailed information, see [Interrupt Handling](#).

In case of a communication failure, you can reset the internal hardware buffers by calling the functions `i2c_reset_tx_fifo()` and `i2c_reset_rx_fifo()` for the send and receive buffers respectively.

Delete Driver When the I2C communication is established with the function `i2c_driver_install()` and is not required for some substantial amount of time, the driver may be deinitialized to release allocated resources by calling `i2c_driver_delete()`.

Application Example

I2C master and slave example: [peripherals/i2c](#).

API Reference

Header File

- `driver/include/driver/i2c.h`

Functions

***esp_err_t* i2c_driver_install** (*i2c_port_t* i2c_num, *i2c_mode_t* mode, *size_t* slv_rx_buf_len, *size_t* slv_tx_buf_len, *int* intr_alloc_flags)

I2C driver install.

Note Only slave mode will use this value, driver will ignore this value in master mode.

Note Only slave mode will use this value, driver will ignore this value in master mode.

Note In master mode, if the cache is likely to be disabled (such as write flash) and the slave is time-sensitive, `ESP_INTR_FLAG_IRAM` is suggested to be used. In this case, please use the memory allocated from internal RAM in i2c read and write function, because we can not access the psram (if psram is enabled) in interrupt handle function when cache is disabled.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_FAIL` Driver install error

Parameters

- `i2c_num`: I2C port number
- `mode`: I2C mode (master or slave)
- `slv_rx_buf_len`: receiving buffer size for slave mode

Parameters

- `slv_tx_buf_len`: sending buffer size for slave mode

Parameters

- `intr_alloc_flags`: Flags used to allocate the interrupt. One or multiple (ORred) `ESP_INTR_FLAG_*` values. See `esp_intr_alloc.h` for more info.

***esp_err_t* i2c_driver_delete** (*i2c_port_t* i2c_num)

I2C driver delete.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `i2c_num`: I2C port number

***esp_err_t* i2c_param_config** (*i2c_port_t* i2c_num, *const* *i2c_config_t* *i2c_conf)

I2C parameter initialization.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `i2c_num`: I2C port number
- `i2c_conf`: pointer to I2C parameter settings

***esp_err_t* i2c_reset_tx_fifo** (*i2c_port_t* i2c_num)

reset I2C tx hardware fifo

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `i2c_num`: I2C port number

***esp_err_t* i2c_reset_rx_fifo** (*i2c_port_t* i2c_num)

reset I2C rx fifo

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `i2c_num`: I2C port number

***esp_err_t* i2c_isr_register** (*i2c_port_t* i2c_num, *void* (*fn) *void* *, *void* *arg, *int* intr_alloc_flags, *intr_handle_t* *handle) I2C isr handler register.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `i2c_num`: I2C port number
- `fn`: isr handler function
- `arg`: parameter for isr handler function
- `intr_alloc_flags`: Flags used to allocate the interrupt. One or multiple (ORred) ESP_INTR_FLAG_* values. See `esp_intr_alloc.h` for more info.
- `handle`: handle return from `esp_intr_alloc`.

esp_err_t **i2c_isr_free** (*intr_handle_t* handle)
to delete and free I2C isr.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `handle`: handle of isr.

esp_err_t **i2c_set_pin** (*i2c_port_t* i2c_num, int sda_io_num, int scl_io_num, bool sda_pullup_en, bool scl_pullup_en, *i2c_mode_t* mode)
Configure GPIO signal for I2C sck and sda.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `i2c_num`: I2C port number
- `sda_io_num`: GPIO number for I2C sda signal
- `scl_io_num`: GPIO number for I2C scl signal
- `sda_pullup_en`: Whether to enable the internal pullup for sda pin
- `scl_pullup_en`: Whether to enable the internal pullup for scl pin
- `mode`: I2C mode

i2c_cmd_handle_t **i2c_cmd_link_create** (void)
Create and init I2C command link.

Note Before we build I2C command link, we need to call `i2c_cmd_link_create()` to create a command link. After we finish sending the commands, we need to call `i2c_cmd_link_delete()` to release and return the resources.

Return i2c command link handler

void **i2c_cmd_link_delete** (*i2c_cmd_handle_t* cmd_handle)
Free I2C command link.

Note Before we build I2C command link, we need to call `i2c_cmd_link_create()` to create a command link. After we finish sending the commands, we need to call `i2c_cmd_link_delete()` to release and return the resources.

Parameters

- `cmd_handle`: I2C command handle

esp_err_t **i2c_master_start** (*i2c_cmd_handle_t* cmd_handle)
Queue command for I2C master to generate a start signal.

Note Only call this function in I2C master mode Call `i2c_master_cmd_begin()` to send all queued commands

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `cmd_handle`: I2C cmd link

esp_err_t **i2c_master_write_byte** (*i2c_cmd_handle_t* cmd_handle, uint8_t data, bool ack_en)
Queue command for I2C master to write one byte to I2C bus.

Note Only call this function in I2C master mode Call `i2c_master_cmd_begin()` to send all queued commands

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `cmd_handle`: I2C cmd link
- `data`: I2C one byte command to write to bus
- `ack_en`: enable ack check for master

`esp_err_t i2c_master_write(i2c_cmd_handle_t cmd_handle, uint8_t *data, size_t data_len, bool ack_en)`

Queue command for I2C master to write buffer to I2C bus.

Note Only call this function in I2C master mode Call `i2c_master_cmd_begin()` to send all queued commands

Note If the psram is enabled and `intr_flag` is `ESP_INTR_FLAG_IRAM`, please use the memory allocated from internal RAM.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `cmd_handle`: I2C cmd link
- `data`: data to send

Parameters

- `data_len`: data length
- `ack_en`: enable ack check for master

`esp_err_t i2c_master_read_byte(i2c_cmd_handle_t cmd_handle, uint8_t *data, i2c_ack_type_t ack)`

Queue command for I2C master to read one byte from I2C bus.

Note Only call this function in I2C master mode Call `i2c_master_cmd_begin()` to send all queued commands

Note If the psram is enabled and `intr_flag` is `ESP_INTR_FLAG_IRAM`, please use the memory allocated from internal RAM.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `cmd_handle`: I2C cmd link
- `data`: pointer accept the data byte

Parameters

- `ack`: ack value for read command

`esp_err_t i2c_master_read(i2c_cmd_handle_t cmd_handle, uint8_t *data, size_t data_len, i2c_ack_type_t ack)`

Queue command for I2C master to read data from I2C bus.

Note Only call this function in I2C master mode Call `i2c_master_cmd_begin()` to send all queued commands

Note If the psram is enabled and `intr_flag` is `ESP_INTR_FLAG_IRAM`, please use the memory allocated from internal RAM.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `cmd_handle`: I2C cmd link
- `data`: data buffer to accept the data from bus

Parameters

- `data_len`: read data length
- `ack`: ack value for read command

`esp_err_t i2c_master_stop(i2c_cmd_handle_t cmd_handle)`

Queue command for I2C master to generate a stop signal.

Note Only call this function in I2C master mode Call `i2c_master_cmd_begin()` to send all queued commands

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- cmd_handle: I2C cmd link

`esp_err_t i2c_master_cmd_begin(i2c_port_t i2c_num, i2c_cmd_handle_t cmd_handle, TickType_t ticks_to_wait)`

I2C master send queued commands. This function will trigger sending all queued commands. The task will be blocked until all the commands have been sent out. The I2C APIs are not thread-safe, if you want to use one I2C port in different tasks, you need to take care of the multi-thread issue.

Note Only call this function in I2C master mode

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Sending command error, slave doesn't ACK the transfer.
- ESP_ERR_INVALID_STATE I2C driver not installed or not in master mode.
- ESP_ERR_TIMEOUT Operation timeout because the bus is busy.

Parameters

- i2c_num: I2C port number
- cmd_handle: I2C command handler
- ticks_to_wait: maximum wait ticks.

`int i2c_slave_write_buffer(i2c_port_t i2c_num, uint8_t *data, int size, TickType_t ticks_to_wait)`

I2C slave write data to internal ringbuffer, when tx fifo empty, isr will fill the hardware fifo from the internal ringbuffer.

Note Only call this function in I2C slave mode

Return

- ESP_FAIL(-1) Parameter error
- Others(>=0) The number of data bytes that pushed to the I2C slave buffer.

Parameters

- i2c_num: I2C port number
- data: data pointer to write into internal buffer
- size: data size
- ticks_to_wait: Maximum waiting ticks

`int i2c_slave_read_buffer(i2c_port_t i2c_num, uint8_t *data, size_t max_size, TickType_t ticks_to_wait)`

I2C slave read data from internal buffer. When I2C slave receive data, isr will copy received data from hardware rx fifo to internal ringbuffer. Then users can read from internal ringbuffer.

Note Only call this function in I2C slave mode

Return

- ESP_FAIL(-1) Parameter error
- Others(>=0) The number of data bytes that read from I2C slave buffer.

Parameters

- i2c_num: I2C port number
- data: data pointer to accept data from internal buffer
- max_size: Maximum data size to read
- ticks_to_wait: Maximum waiting ticks

`esp_err_t i2c_set_period(i2c_port_t i2c_num, int high_period, int low_period)`

set I2C master clock period

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- i2c_num: I2C port number
- high_period: clock cycle number during SCL is high level, high_period is a 14 bit value
- low_period: clock cycle number during SCL is low level, low_period is a 14 bit value

esp_err_t **i2c_get_period** (*i2c_port_t* *i2c_num*, int **high_period*, int **low_period*)
get I2C master clock period

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- *i2c_num*: I2C port number
- *high_period*: pointer to get clock cycle number during SCL is high level, will get a 14 bit value
- *low_period*: pointer to get clock cycle number during SCL is low level, will get a 14 bit value

esp_err_t **i2c_filter_enable** (*i2c_port_t* *i2c_num*, uint8_t *cyc_num*)

enable hardware filter on I2C bus Sometimes the I2C bus is disturbed by high frequency noise(about 20ns), or the rising edge of the SCL clock is very slow, these may cause the master state machine broken. enable hardware filter can filter out high frequency interference and make the master more stable.

Note Enable filter will slow the SCL clock.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- *i2c_num*: I2C port number
- *cyc_num*: the APB cycles need to be filtered($0 \leq cyc_num \leq 7$). When the period of a pulse is less than $cyc_num * APB_cycle$, the I2C controller will ignore this pulse.

esp_err_t **i2c_filter_disable** (*i2c_port_t* *i2c_num*)

disable filter on I2C bus

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- *i2c_num*: I2C port number

esp_err_t **i2c_set_start_timing** (*i2c_port_t* *i2c_num*, int *setup_time*, int *hold_time*)

set I2C master start signal timing

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- *i2c_num*: I2C port number
- *setup_time*: clock number between the falling-edge of SDA and rising-edge of SCL for start mark, it' s a 10-bit value.
- *hold_time*: clock num between the falling-edge of SDA and falling-edge of SCL for start mark, it' s a 10-bit value.

esp_err_t **i2c_get_start_timing** (*i2c_port_t* *i2c_num*, int **setup_time*, int **hold_time*)

get I2C master start signal timing

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- *i2c_num*: I2C port number
- *setup_time*: pointer to get setup time
- *hold_time*: pointer to get hold time

esp_err_t **i2c_set_stop_timing** (*i2c_port_t* *i2c_num*, int *setup_time*, int *hold_time*)

set I2C master stop signal timing

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `i2c_num`: I2C port number
- `setup_time`: clock num between the rising-edge of SCL and the rising-edge of SDA, it' s a 10-bit value.
- `hold_time`: clock number after the STOP bit' s rising-edge, it' s a 14-bit value.

`esp_err_t i2c_get_stop_timing (i2c_port_t i2c_num, int *setup_time, int *hold_time)`

get I2C master stop signal timing

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `i2c_num`: I2C port number
- `setup_time`: pointer to get setup time.
- `hold_time`: pointer to get hold time.

`esp_err_t i2c_set_data_timing (i2c_port_t i2c_num, int sample_time, int hold_time)`

set I2C data signal timing

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `i2c_num`: I2C port number
- `sample_time`: clock number I2C used to sample data on SDA after the rising-edge of SCL, it' s a 10-bit value
- `hold_time`: clock number I2C used to hold the data after the falling-edge of SCL, it' s a 10-bit value

`esp_err_t i2c_get_data_timing (i2c_port_t i2c_num, int *sample_time, int *hold_time)`

get I2C data signal timing

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `i2c_num`: I2C port number
- `sample_time`: pointer to get sample time
- `hold_time`: pointer to get hold time

`esp_err_t i2c_set_timeout (i2c_port_t i2c_num, int timeout)`

set I2C timeout value

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `i2c_num`: I2C port number
- `timeout`: timeout value for I2C bus (unit: APB 80Mhz clock cycle)

`esp_err_t i2c_get_timeout (i2c_port_t i2c_num, int *timeout)`

get I2C timeout value

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `i2c_num`: I2C port number
- `timeout`: pointer to get timeout value

`esp_err_t i2c_set_data_mode (i2c_port_t i2c_num, i2c_trans_mode_t tx_trans_mode, i2c_trans_mode_t rx_trans_mode)`

set I2C data transfer mode

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `i2c_num`: I2C port number
- `tx_trans_mode`: I2C sending data mode
- `rx_trans_mode`: I2C receiving data mode

`esp_err_t i2c_get_data_mode(i2c_port_t i2c_num, i2c_trans_mode_t *tx_trans_mode, i2c_trans_mode_t *rx_trans_mode)`

get I2C data transfer mode

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `i2c_num`: I2C port number
- `tx_trans_mode`: pointer to get I2C sending data mode
- `rx_trans_mode`: pointer to get I2C receiving data mode

Macros**I2C_APB_CLK_FREQ**

I2C source clock is APB clock, 80MHz

I2C_NUM_0

I2C port 0

I2C_NUM_1

I2C port 1

I2C_NUM_MAX

I2C port max

Type Definitions

`typedef void *i2c_cmd_handle_t`

I2C command handle

Header File

- [soc/include/hal/i2c_types.h](#)

Structures

`struct i2c_config_t`

I2C initialization parameters.

Public Members

`i2c_mode_t mode`

I2C mode

int `sda_io_num`

GPIO number for I2C sda signal

int `scl_io_num`

GPIO number for I2C scl signal

bool `sda_pullup_en`

Internal GPIO pull mode for I2C sda signal

bool **scl_pullup_en**
Internal GPIO pull mode for I2C scl signal

uint32_t **clk_speed**
I2C clock frequency for master mode, (no higher than 1MHz for now)

uint8_t **addr_10bit_en**
I2C 10bit address mode enable for slave mode

uint16_t **slave_addr**
I2C address for slave mode

Type Definitions

typedef int i2c_port_t
I2C port number, can be I2C_NUM_0 ~ (I2C_NUM_MAX-1).

Enumerations

enum i2c_mode_t
Values:

I2C_MODE_SLAVE = 0
I2C slave mode

I2C_MODE_MASTER
I2C master mode

I2C_MODE_MAX

enum i2c_rw_t
Values:

I2C_MASTER_WRITE = 0
I2C write data

I2C_MASTER_READ
I2C read data

enum i2c_opmode_t
Values:

I2C_CMD_RESTART = 0
I2C restart command

I2C_CMD_WRITE
I2C write command

I2C_CMD_READ
I2C read command

I2C_CMD_STOP
I2C stop command

I2C_CMD_END
I2C end command

enum i2c_trans_mode_t
Values:

I2C_DATA_MODE_MSB_FIRST = 0
I2C data msb first

I2C_DATA_MODE_LSB_FIRST = 1
I2C data lsb first

I2C_DATA_MODE_MAX

enum i2c_addr_mode_t

Values:

I2C_ADDR_BIT_7 = 0
I2C 7bit address for slave mode

I2C_ADDR_BIT_10
I2C 10bit address for slave mode

I2C_ADDR_BIT_MAX

enum i2c_ack_type_t

Values:

I2C_MASTER_ACK = 0x0
I2C ack for each byte read

I2C_MASTER_NACK = 0x1
I2C nack for each byte read

I2C_MASTER_LAST_NACK = 0x2
I2C nack for the last byte

I2C_MASTER_ACK_MAX

enum i2c_sclk_t

Values:

I2C_SCLK_REF_TICK
I2C source clock from REF_TICK

I2C_SCLK_APB
I2C source clock from APB

2.3.6 I2S

Overview

I2S (Inter-IC Sound) is a serial, synchronous communication protocol that is usually used for transmitting audio data between two digital audio devices.

ESP32 integrates two I2S controllers, referred to as I2S0 and I2S1, both of which can be used for streaming audio and video digital data.

An I2S bus consists of the following lines:

- Bit clock line
- Channel select line
- Serial data line

Each I2S controller has the following features that can be configured using the I2S driver:

- Operation as system master or slave
- Capable of acting as transmitter or receiver
- Dedicated DMA controller that allows for streaming sample data without requiring the CPU to copy each data sample

Each controller can operate in half-duplex communication mode. Thus, the two controllers can be combined to establish full-duplex communication.

I2S0 output can be routed directly to the digital-to-analog converter's (DAC) output channels (GPIO 25 & GPIO 26) to produce direct analog output without involving any external I2S codecs. I2S0 can also be used for transmitting PDM (Pulse-density modulation) signals.

The I2S peripherals also support LCD mode for communicating data over a parallel bus, as used by some LCD displays and camera modules. LCD mode has the following operational modes:

- LCD master transmitting mode
- Camera slave receiving mode
- ADC/DAC mode

For more information, see the [ESP32 Technical Reference Manual](#).

Note: For high accuracy clock applications, use the APLL_CLK clock source, which has the frequency range of 16 ~ 128 MHz. You can enable the APLL_CLK clock source by setting `i2s_config_t::use_apll` to TRUE.

If `i2s_config_t::use_apll` = TRUE and `i2s_config_t::fixed_mclk` > 0, then the master clock output frequency for I2S will be equal to the value of `i2s_config_t::fixed_mclk`, which means that the mclk frequency is provided by the user, instead of being calculated by the driver.

The clock rate of the word select line, which is called audio left-right clock rate (LRCK) here, is always the divisor of the master clock output frequency and for which the following is always true: $0 < \text{MCLK/LRCK/channels/bits_per_sample} < 64$.

Functional Overview

Installing the Driver Install the I2S driver by calling the function `:cpp:func`i2s_driver_install`` and passing the following arguments:

- Port number
- The structure `i2s_config_t` with defined communication parameters
- Event queue size and handle

Configuration example:

```
static const int i2s_num = 0; // i2s port number

static const i2s_config_t i2s_config = {
    .mode = I2S_MODE_MASTER | I2S_MODE_TX,
    .sample_rate = 44100,
    .bits_per_sample = 16,
    .channel_format = I2S_CHANNEL_FMT_RIGHT_LEFT,
    .communication_format = I2S_COMM_FORMAT_I2S | I2S_COMM_FORMAT_I2S_MSB,
    .intr_alloc_flags = 0, // default interrupt priority
    .dma_buf_count = 8,
    .dma_buf_len = 64,
    .use_apll = false
};

i2s_driver_install(I2S_NUM, &i2s_config, 0, NULL);
```

Setting Communication Pins Once the driver is installed, configure physical GPIO pins to which signals will be routed. For this, call the function `:cpp:func`i2s_set_pin`` and pass the following arguments to it:

- Port number
- The structure `i2s_pin_config_t` defining the GPIO pin numbers to which the driver should route the BCK, WS, DATA out, and DATA in signals. If you want to keep a currently allocated pin number for a specific signal, or if this signal is unused, then pass the macro `I2S_PIN_NO_CHANGE`. See the example below.

```
static const i2s_pin_config_t pin_config = {
    .bck_io_num = 26,
    .ws_io_num = 25,
    .data_out_num = 22,
    .data_in_num = I2S_PIN_NO_CHANGE
};
```

(continues on next page)

```
i2s_set_pin(i2s_num, &pin_config);
```

Running I2S Communication To perform a transmission:

- Prepare the data for sending
- Call the function `i2s_write()` and pass the data buffer address and data length to it

The function will write the data to the I2S DMA Tx buffer, and then the data will be transmitted automatically.

```
i2s_write(I2S_NUM, samples_data, ((bits+8)/16)*SAMPLE_PER_CYCLE*4, &i2s_bytes_
↪write, 100);
```

To retrieve received data, use the function `i2s_read()`. It will retrieve the data from the I2S DMA Rx buffer, once the data is received by the I2S controller.

You can temporarily stop the I2S driver by calling the function `i2s_stop()`, which will disable the I2S Tx/Rx units until the function `i2s_start()` is called. If the function `cpp:func:i2s_driver_install` is used, the driver will start up automatically eliminating the need to call `i2s_start()`.

Deleting the Driver If the established communication is no longer required, the driver can be removed to free allocated resources by calling `i2s_driver_uninstall()`.

Application Example

A code example for the I2S driver can be found in the directory [peripherals/i2s](#).

In addition, there are two short configuration examples for the I2S driver.

I2S configuration

```
#include "driver/i2s.h"
#include "freertos/queue.h"

static const int i2s_num = 0; // i2s port number

static const i2s_config_t i2s_config = {
    .mode = I2S_MODE_MASTER | I2S_MODE_TX,
    .sample_rate = 44100,
    .bits_per_sample = 16,
    .channel_format = I2S_CHANNEL_FMT_RIGHT_LEFT,
    .communication_format = I2S_COMM_FORMAT_I2S | I2S_COMM_FORMAT_I2S_MSB,
    .intr_alloc_flags = 0, // default interrupt priority
    .dma_buf_count = 8,
    .dma_buf_len = 64,
    .use_apll = false
};

static const i2s_pin_config_t pin_config = {
    .bck_io_num = 26,
    .ws_io_num = 25,
    .data_out_num = 22,
    .data_in_num = I2S_PIN_NO_CHANGE
};

...

i2s_driver_install(i2s_num, &i2s_config, 0, NULL); //install and start i2s_
↪driver
```

(continues on next page)

(continued from previous page)

```

i2s_set_pin(i2s_num, &pin_config);

i2s_set_sample_rates(i2s_num, 22050); //set sample rates

i2s_driver_uninstall(i2s_num); //stop & destroy i2s driver

```

Configuring I2S to use internal DAC for analog output

```

#include "driver/i2s.h"
#include "freertos/queue.h"

static const int i2s_num = 0; // i2s port number

static const i2s_config_t i2s_config = {
    .mode = I2S_MODE_MASTER | I2S_MODE_TX | I2S_MODE_DAC_BUILT_IN,
    .sample_rate = 44100,
    .bits_per_sample = 16, /* the DAC module will only take the 8bits from MSB */
    .channel_format = I2S_CHANNEL_FMT_RIGHT_LEFT,
    .communication_format = I2S_COMM_FORMAT_I2S_MSB,
    .intr_alloc_flags = 0, // default interrupt priority
    .dma_buf_count = 8,
    .dma_buf_len = 64,
    .use_apll = false
};

...

i2s_driver_install(i2s_num, &i2s_config, 0, NULL); //install and start i2s_
↳driver

i2s_set_pin(i2s_num, NULL); //for internal DAC, this will enable both of the_
↳internal channels

//You can call i2s_set_dac_mode to set built-in DAC output mode.
//i2s_set_dac_mode(I2S_DAC_CHANNEL_BOTH_EN);

i2s_set_sample_rates(i2s_num, 22050); //set sample rates

i2s_driver_uninstall(i2s_num); //stop & destroy i2s driver

```

API Reference

Header File

- [driver/include/driver/i2s.h](#)

Functions

esp_err_t **i2s_set_pin** (*i2s_port_t* i2s_num, **const** *i2s_pin_config_t* *pin)

Set I2S pin number.

Inside the pin configuration structure, set I2S_PIN_NO_CHANGE for any pin where the current configuration should not be changed.

Note The I2S peripheral output signals can be connected to multiple GPIO pads. However, the I2S peripheral input signal can only be connected to one GPIO pad.

Parameters

- i2s_num: I2S_NUM_0 or I2S_NUM_1

- `pin`: I2S Pin structure, or NULL to set 2-channel 8-bit internal DAC pin configuration (GPIO25 & GPIO26)

Note if `*pin` is set as NULL, this function will initialize both of the built-in DAC channels by default. if you don't want this to happen and you want to initialize only one of the DAC channels, you can call `i2s_set_dac_mode` instead.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL IO error

esp_err_t **i2s_set_dac_mode** (*i2s_dac_mode_t* `dac_mode`)

Set I2S dac mode, I2S built-in DAC is disabled by default.

Note Built-in DAC functions are only supported on I2S0 for current ESP32 chip. If either of the built-in DAC channel are enabled, the other one can not be used as RTC DAC function at the same time.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `dac_mode`: DAC mode configurations - see `i2s_dac_mode_t`

esp_err_t **i2s_driver_install** (*i2s_port_t* `i2s_num`, **const** *i2s_config_t* `*i2s_config`, `int` `queue_size`, `void` `*i2s_queue`)

Install and start I2S driver.

This function must be called before any I2S driver read/write operations.

Parameters

- `i2s_num`: I2S_NUM_0, I2S_NUM_1
- `i2s_config`: I2S configurations - see `i2s_config_t` struct
- `queue_size`: I2S event queue size/depth.
- `i2s_queue`: I2S event queue handle, if set NULL, driver will not use an event queue.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_NO_MEM Out of memory

esp_err_t **i2s_driver_uninstall** (*i2s_port_t* `i2s_num`)

Uninstall I2S driver.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `i2s_num`: I2S_NUM_0, I2S_NUM_1

esp_err_t **i2s_write** (*i2s_port_t* `i2s_num`, **const** `void` `*src`, `size_t` `size`, `size_t` `*bytes_written`, `TickType_t` `ticks_to_wait`)

Write data to I2S DMA transmit buffer.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `i2s_num`: I2S_NUM_0, I2S_NUM_1
- `src`: Source address to write from
- `size`: Size of data in bytes
- `[out] bytes_written`: Number of bytes written, if timeout, the result will be less than the size passed in.
- `ticks_to_wait`: TX buffer wait timeout in RTOS ticks. If this many ticks pass without space becoming available in the DMA transmit buffer, then the function will return (note that if the data

is written to the DMA buffer in pieces, the overall operation may still take longer than this timeout.)
Pass portMAX_DELAY for no timeout.

esp_err_t **i2s_write_expand** (*i2s_port_t* *i2s_num*, const void **src*, size_t *size*, size_t *src_bits*, size_t *aim_bits*, size_t **bytes_written*, TickType_t *ticks_to_wait*)

Write data to I2S DMA transmit buffer while expanding the number of bits per sample. For example, expanding 16-bit PCM to 32-bit PCM.

Format of the data in source buffer is determined by the I2S configuration (see *i2s_config_t*).

Parameters

- *i2s_num*: I2S_NUM_0, I2S_NUM_1
- *src*: Source address to write from
- *size*: Size of data in bytes
- *src_bits*: Source audio bit
- *aim_bits*: Bit wanted, no more than 32, and must be greater than *src_bits*
- [out] *bytes_written*: Number of bytes written, if timeout, the result will be less than the size passed in.
- *ticks_to_wait*: TX buffer wait timeout in RTOS ticks. If this many ticks pass without space becoming available in the DMA transmit buffer, then the function will return (note that if the data is written to the DMA buffer in pieces, the overall operation may still take longer than this timeout.) Pass portMAX_DELAY for no timeout.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **i2s_read** (*i2s_port_t* *i2s_num*, void **dest*, size_t *size*, size_t **bytes_read*, TickType_t *ticks_to_wait*)

Read data from I2S DMA receive buffer.

Note If the built-in ADC mode is enabled, we should call *i2s_adc_start* and *i2s_adc_stop* around the whole reading process, to prevent the data getting corrupted.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- *i2s_num*: I2S_NUM_0, I2S_NUM_1
- *dest*: Destination address to read into
- *size*: Size of data in bytes
- [out] *bytes_read*: Number of bytes read, if timeout, bytes read will be less than the size passed in.
- *ticks_to_wait*: RX buffer wait timeout in RTOS ticks. If this many ticks pass without bytes becoming available in the DMA receive buffer, then the function will return (note that if data is read from the DMA buffer in pieces, the overall operation may still take longer than this timeout.) Pass portMAX_DELAY for no timeout.

esp_err_t **i2s_set_sample_rates** (*i2s_port_t* *i2s_num*, uint32_t *rate*)

Set sample rate used for I2S RX and TX.

The bit clock rate is determined by the sample rate and *i2s_config_t* configuration parameters (number of channels, *bits_per_sample*).

$\text{bit_clock} = \text{rate} * (\text{number of channels}) * \text{bits_per_sample}$

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_NO_MEM Out of memory

Parameters

- *i2s_num*: I2S_NUM_0, I2S_NUM_1
- *rate*: I2S sample rate (ex: 8000, 44100...)

esp_err_t **i2s_stop** (*i2s_port_t* *i2s_num*)

Stop I2S driver.

There is no need to call `i2s_stop()` before calling `i2s_driver_uninstall()`.

Disables I2S TX/RX, until `i2s_start()` is called.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- *i2s_num*: I2S_NUM_0, I2S_NUM_1

esp_err_t **i2s_start** (*i2s_port_t* *i2s_num*)

Start I2S driver.

It is not necessary to call this function after `i2s_driver_install()` (it is started automatically), however it is necessary to call it after `i2s_stop()`.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- *i2s_num*: I2S_NUM_0, I2S_NUM_1

esp_err_t **i2s_zero_dma_buffer** (*i2s_port_t* *i2s_num*)

Zero the contents of the TX DMA buffer.

Pushes zero-byte samples into the TX DMA buffer, until it is full.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- *i2s_num*: I2S_NUM_0, I2S_NUM_1

esp_err_t **i2s_set_clk** (*i2s_port_t* *i2s_num*, *uint32_t* *rate*, *i2s_bits_per_sample_t* *bits*, *i2s_channel_t* *ch*)

Set clock & bit width used for I2S RX and TX.

Similar to `i2s_set_sample_rates()`, but also sets bit width.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_NO_MEM Out of memory

Parameters

- *i2s_num*: I2S_NUM_0, I2S_NUM_1
- *rate*: I2S sample rate (ex: 8000, 44100...)
- *bits*: I2S bit width (I2S_BITS_PER_SAMPLE_16BIT, I2S_BITS_PER_SAMPLE_24BIT, I2S_BITS_PER_SAMPLE_32BIT)
- *ch*: I2S channel, (I2S_CHANNEL_MONO, I2S_CHANNEL_STEREO)

float **i2s_get_clk** (*i2s_port_t* *i2s_num*)

get clock set on particular port number.

Return

- actual clock set by i2s driver

Parameters

- *i2s_num*: I2S_NUM_0, I2S_NUM_1

esp_err_t **i2s_set_adc_mode** (*adc_unit_t* *adc_unit*, *adc1_channel_t* *adc_channel*)

Set built-in ADC mode for I2S DMA, this function will initialize ADC pad, and set ADC parameters.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `adc_unit`: SAR ADC unit index
- `adc_channel`: ADC channel index

esp_err_t **i2s_adc_enable** (*i2s_port_t* *i2s_num*)

Start to use I2S built-in ADC mode.

Note This function would acquire the lock of ADC to prevent the data getting corrupted during the I2S peripheral is being used to do fully continuous ADC sampling.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_ERR_INVALID_STATE` Driver state error

Parameters

- `i2s_num`: i2s port index

esp_err_t **i2s_adc_disable** (*i2s_port_t* *i2s_num*)

Stop to use I2S built-in ADC mode.

Note This function would release the lock of ADC so that other tasks can use ADC.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_ERR_INVALID_STATE` Driver state error

Parameters

- `i2s_num`: i2s port index

Type Definitions

typedef *intr_handle_t* **i2s_isr_handle_t**

Header File

- `soc/include/hal/i2s_types.h`

Structures

struct **i2s_config_t**

I2S configuration parameters for `i2s_param_config` function.

Public Members

i2s_mode_t **mode**

I2S work mode

int **sample_rate**

I2S sample rate

i2s_bits_per_sample_t **bits_per_sample**

I2S bits per sample

i2s_channel_fmt_t **channel_format**

I2S channel format

i2s_comm_format_t **communication_format**

I2S communication format

int **intr_alloc_flags**

Flags used to allocate the interrupt. One or multiple (ORred) `ESP_INTR_FLAG_*` values. See `esp_intr_alloc.h` for more info

int **dma_buf_count**

I2S DMA Buffer Count

int **dma_buf_len**
I2S DMA Buffer Length

bool **use_apll**
I2S using APLL as main I2S clock, enable it to get accurate clock

bool **tx_desc_auto_clear**
I2S auto clear tx descriptor if there is underflow condition (helps in avoiding noise in case of data unavailability)

int **fixed_mclk**
I2S using fixed MCLK output. If use_apll = true and fixed_mclk > 0, then the clock output for i2s is fixed and equal to the fixed_mclk value.

struct i2s_event_t
Event structure used in I2S event queue.

Public Members

i2s_event_type_t **type**
I2S event type

size_t **size**
I2S data size for I2S_DATA event

struct i2s_pin_config_t
I2S pin number for i2s_set_pin.

Public Members

int **bck_io_num**
BCK in out pin

int **ws_io_num**
WS in out pin

int **data_out_num**
DATA out pin

int **data_in_num**
DATA in pin

Macros

I2S_PIN_NO_CHANGE
Use in *i2s_pin_config_t* for pins which should not be changed

Type Definitions

typedef int i2s_port_t
I2S port number, the max port number is (I2S_NUM_MAX -1).

Enumerations

enum i2s_bits_per_sample_t
I2S bit width per sample.

Values:

I2S_BITS_PER_SAMPLE_8BIT = 8
I2S bits per sample: 8-bits

I2S_BITS_PER_SAMPLE_16BIT = 16
I2S bits per sample: 16-bits

I2S_BITS_PER_SAMPLE_24BIT = 24
I2S bits per sample: 24-bits

I2S_BITS_PER_SAMPLE_32BIT = 32
I2S bits per sample: 32-bits

enum i2s_channel_t
I2S channel.

Values:

I2S_CHANNEL_MONO = 1
I2S 1 channel (mono)

I2S_CHANNEL_STEREO = 2
I2S 2 channel (stereo)

enum i2s_comm_format_t
I2S communication standard format.

Values:

I2S_COMM_FORMAT_I2S = 0x01
I2S communication format I2S

I2S_COMM_FORMAT_I2S_MSB = 0x02
I2S format MSB

I2S_COMM_FORMAT_I2S_LSB = 0x04
I2S format LSB

I2S_COMM_FORMAT_PCM = 0x08
I2S communication format PCM

I2S_COMM_FORMAT_PCM_SHORT = 0x10
PCM Short

I2S_COMM_FORMAT_PCM_LONG = 0x20
PCM Long

enum i2s_channel_fmt_t
I2S channel format type.

Values:

I2S_CHANNEL_FMT_RIGHT_LEFT = 0x00

I2S_CHANNEL_FMT_ALL_RIGHT

I2S_CHANNEL_FMT_ALL_LEFT

I2S_CHANNEL_FMT_ONLY_RIGHT

I2S_CHANNEL_FMT_ONLY_LEFT

enum pdm_sample_rate_ratio_t
PDM sample rate ratio, measured in Hz.

Values:

PDM_SAMPLE_RATE_RATIO_64

PDM_SAMPLE_RATE_RATIO_128

enum pdm_pcm_conv_t
PDM PCM convter enable/disable.

Values:

PDM_PCM_CONV_ENABLE

PDM_PCM_CONV_DISABLE

enum i2s_mode_t

I2S Mode, default is I2S_MODE_MASTER | I2S_MODE_TX.

Note PDM and built-in DAC functions are only supported on I2S0 for current ESP32 chip.

Values:

I2S_MODE_MASTER = 1

I2S_MODE_SLAVE = 2

I2S_MODE_TX = 4

I2S_MODE_RX = 8

I2S_MODE_DAC_BUILT_IN = 16

Output I2S data to built-in DAC, no matter the data format is 16bit or 32 bit, the DAC module will only take the 8bits from MSB

I2S_MODE_ADC_BUILT_IN = 32

Input I2S data from built-in ADC, each data can be 12-bit width at most

enum i2s_event_type_t

I2S event types.

Values:

I2S_EVENT_DMA_ERROR

I2S_EVENT_TX_DONE

I2S DMA finish sent 1 buffer

I2S_EVENT_RX_DONE

I2S DMA finish received 1 buffer

I2S_EVENT_MAX

I2S event max index

enum i2s_dac_mode_t

I2S DAC mode for i2s_set_dac_mode.

Note PDM and built-in DAC functions are only supported on I2S0 for current ESP32 chip.

Values:

I2S_DAC_CHANNEL_DISABLE = 0

Disable I2S built-in DAC signals

I2S_DAC_CHANNEL_RIGHT_EN = 1

Enable I2S built-in DAC right channel, maps to DAC channel 1 on GPIO25

I2S_DAC_CHANNEL_LEFT_EN = 2

Enable I2S built-in DAC left channel, maps to DAC channel 2 on GPIO26

I2S_DAC_CHANNEL_BOTH_EN = 0x3

Enable both of the I2S built-in DAC channels.

I2S_DAC_CHANNEL_MAX = 0x4

I2S built-in DAC mode max index

enum i2s_clock_src_t

Values:

I2S_CLK_D2CLK = 0

I2S_CLK_APLL

2.3.7 LED Control

Introduction

The LED control (LEDC) peripheral is primarily designed to control the intensity of LEDs, although it can also be used to generate PWM signals for other purposes as well. It has 16 channels which can generate independent waveforms that can be used, for example, to drive RGB LED devices.

A half of LEDC's channels operate in high speed mode. This mode is implemented in hardware and offers automatic and glitch-free changing of the PWM duty cycle. The other half of channels operate in low speed mode, where the moment of change depends on the application software. Each group of channels is also able to use different clock sources, but this feature is not yet supported in the LEDC driver.

The PWM controller can automatically increase or decrease the duty cycle gradually, allowing for fades without any processor interference.

Functionality Overview

Getting LEDC to work on a specific channel in either *high or low speed mode* is done in three steps:

1. *Configure Timer* by specifying the PWM signal's frequency and duty cycle resolution.
2. *Configure Channel* by associating it with the timer and GPIO to output the PWM signal.
3. *Change PWM Signal* that drives the output in order to change LED's intensity. This can be done under the full control of software or with hardware fading functions.

As an optional step, it is also possible to set up an interrupt on the fade end.

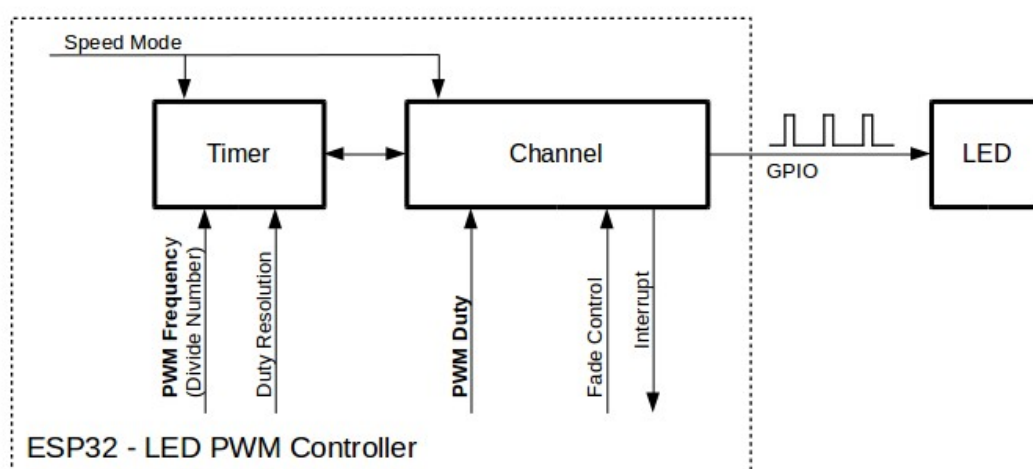


Fig. 14: Key Settings of LED PWM Controller's API

Configure Timer Setting the timer is done by calling the function `ledc_timer_config()` and passing to it a data structure `ledc_timer_config_t` that contains the following configuration settings:

- Timer number `ledc_timer_t`
- Speed mode `ledc_mode_t`
- PWM signal frequency
- Resolution of PWM duty

The frequency and the duty resolution are interdependent. The higher the PWM frequency, the lower duty resolution is available, and vice versa. This relationship might be important if you are planning to use this API for purposes

other than changing the intensity of LEDs. For more details, see Section [Supported Range of Frequency and Duty Resolutions](#).

Configure Channel When the timer is set up, configure a selected channel (one out of `ledc_channel_t`). This is done by calling the function `ledc_channel_config()`.

Similar to the timer configuration, the channel setup function should be passed a structure `ledc_channel_config_t` that contains the channel's configuration parameters.

At this point, the channel should start operating and generating the PWM signal on the selected GPIO, as configured in `ledc_channel_config_t`, with the frequency specified in the timer settings and the given duty cycle. The channel operation (signal generation) can be suspended at any time by calling the function `ledc_stop()`.

Change PWM Signal Once the channel starts operating and generating the PWM signal with the constant duty cycle and frequency, there are a couple of ways to change this signal. When driving LEDs, primarily the duty cycle is changed to vary the light intensity.

The following two sections describe how to change the duty cycle using software and hardware fading. If required, the signal's frequency can also be changed; it is covered in Section [Change PWM Frequency](#).

Change PWM Duty Cycle Using Software To set the duty cycle, use the dedicated function `ledc_set_duty()`. After that, call `ledc_update_duty()` to activate the changes. To check the currently set value, use the corresponding `_get_` function `ledc_get_duty()`.

Another way to set the duty cycle, as well as some other channel parameters, is by calling `ledc_channel_config()` covered in Section [Configure Channel](#).

The range of the duty cycle values passed to functions depends on selected `duty_resolution` and should be from 0 to $(2 ** \text{duty_resolution}) - 1$. For example, if the selected duty resolution is 10, then the duty cycle values can range from 0 to 1023. This provides the resolution of ~0.1%.

Change PWM Duty Cycle using Hardware The LEDC hardware provides the means to gradually transition from one duty cycle value to another. To use this functionality, enable fading with `ledc_fade_func_install()` and then configure it by calling one of the available fading functions:

- `ledc_set_fade_with_time()`
- `ledc_set_fade_with_step()`
- `ledc_set_fade()`

Finally start fading with `ledc_fade_start()`.

If not required anymore, fading and an associated interrupt can be disabled with `ledc_fade_func_uninstall()`.

Change PWM Frequency The LEDC API provides several ways to change the PWM frequency “on the fly” :

- Set the frequency by calling `ledc_set_freq()`. There is a corresponding function `ledc_get_freq()` to check the current frequency.
- Change the frequency and the duty resolution by calling `ledc_bind_channel_timer()` to bind some other timer to the channel.
- Change the channel's timer by calling `ledc_channel_config()`.

More Control Over PWM There are several lower level timer-specific functions that can be used to change PWM settings:

- `ledc_timer_set()`
- `ledc_timer_rst()`
- `ledc_timer_pause()`
- `ledc_timer_resume()`

The first two functions are called “behind the scenes” by `ledc_channel_config()` to provide a “clean” startup of a timer after it is configured.

Use Interrupts When configuring an LEDC channel, one of the parameters selected within `ledc_channel_config_t` is `ledc_intr_type_t` which triggers an interrupt on fade completion.

For registration of a handler to address this interrupt, call `ledc_isr_register()`.

LEDC High and Low Speed Mode

Of the total 8 timers and 16 channels available in the LED PWM Controller, half of them are dedicated to operation in high speed mode and the other half in low speed mode. Selection of a low or high speed timer or channel is done with the parameter `ledc_mode_t` that can be found in applicable function calls.

The advantage of high speed mode is hardware-supported, glitch-free changeover of the timer settings. This means that if the timer settings are modified, the changes will be applied automatically on the next overflow interrupt of the timer. In contrast, when updating the low-speed timer, the change of settings should be explicitly triggered by software. The LEDC driver handles it in the background, e.g., when `ledc_timer_config()` or `ledc_timer_set()` is called.

For additional details regarding speed modes, refer to [ESP32 Technical Reference Manual \(PDF\)](#). Please note that the support for `SLOW_CLOCK` mentioned in this manual is not yet supported in the LEDC driver.

Supported Range of Frequency and Duty Resolutions

The LED PWM Controller is designed primarily to drive LEDs. It provides a wide resolution for PWM duty cycle settings. For instance, the PWM frequency of 5 kHz can have the maximum duty resolution of 13 bits. It means that the duty can be set anywhere from 0 to 100% with a resolution of ~0.012% ($2^{13} = 8192$ discrete levels of the LED intensity).

The LEDC can be used for generating signals at much higher frequencies that are sufficient enough to clock other devices, e.g., a digital camera module. In this case, the maximum available frequency is 40 MHz with duty resolution of 1 bit. This means that the duty cycle is fixed at 50% and cannot be adjusted.

The LEDC API is designed to report an error when trying to set a frequency and a duty resolution that exceed the range of LEDC’s hardware. For example, an attempt to set the frequency to 20 MHz and the duty resolution to 3 bits will result in the following error reported on a serial monitor:

```
E (196) ledc: requested frequency and duty resolution cannot be achieved, try_
↪reducing freq_hz or duty_resolution. div_param=128
```

In such a situation, either the duty resolution or the frequency must be reduced. For example, setting the duty resolution to 2 will resolve this issue and will make it possible to set the duty cycle at 25% steps, i.e., at 25%, 50% or 75%.

The LEDC driver will also capture and report attempts to configure frequency / duty resolution combinations that are below the supported minimum, e.g.:

```
E (196) ledc: requested frequency and duty resolution cannot be achieved, try_
↪increasing freq_hz or duty_resolution. div_param=128000000
```

The duty resolution is normally set using `ledc_timer_bit_t`. This enumeration covers the range from 10 to 15 bits. If a smaller duty resolution is required (from 10 down to 1), enter the equivalent numeric values directly.

Application Example

The LEDC change duty cycle and fading control example: [peripherals/ledc](#).

API Reference

Header File

- [driver/include/driver/ledc.h](#)

Functions

esp_err_t **ledc_channel_config**(const *ledc_channel_config_t* *ledc_conf)

LEDC channel configuration Configure LEDC channel with the given channel/output gpio_num/interrupt/source timer/frequency(Hz)/LEDC duty resolution.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- ledc_conf: Pointer of LEDC channel configure struct

esp_err_t **ledc_timer_config**(const *ledc_timer_config_t* *timer_conf)

LEDC timer configuration Configure LEDC timer with the given source timer/frequency(Hz)/duty_resolution.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Can not find a proper pre-divider number base on the given frequency and the current duty_resolution.

Parameters

- timer_conf: Pointer of LEDC timer configure struct

esp_err_t **ledc_update_duty**(*ledc_mode_t* speed_mode, *ledc_channel_t* channel)

LEDC update channel parameters.

Note Call this function to activate the LEDC updated parameters. After ledc_set_duty, we need to call this function to update the settings. And the new LEDC parameters don't take effect until the next PWM cycle.

Note ledc_set_duty, ledc_set_duty_with_hpoint and ledc_update_duty are not thread-safe, do not call these functions to control one LEDC channel in different tasks at the same time. A thread-safe version of API is ledc_set_duty_and_update

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- speed_mode: Select the LEDC speed_mode, high-speed mode and low-speed mode,
- channel: LEDC channel (0-7), select from ledc_channel_t

esp_err_t **ledc_set_pin**(int gpio_num, *ledc_mode_t* speed_mode, *ledc_channel_t* ledc_channel)

Set LEDC output gpio.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- gpio_num: The LEDC output gpio
- speed_mode: Select the LEDC speed_mode, high-speed mode and low-speed mode
- ledc_channel: LEDC channel (0-7), select from ledc_channel_t

esp_err_t **ledc_stop**(*ledc_mode_t* speed_mode, *ledc_channel_t* channel, uint32_t idle_level)

LEDC stop. Disable LEDC output, and set idle level.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- speed_mode: Select the LEDC speed_mode, high-speed mode and low-speed mode

- `channel`: LEDC channel (0-7), select from `ledc_channel_t`
- `idle_level`: Set output idle level after LEDC stops.

`esp_err_t ledc_set_freq` (*ledc_mode_t speed_mode, ledc_timer_t timer_num, uint32_t freq_hz*)

LEDC set channel frequency (Hz)

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_FAIL` Can not find a proper pre-divider number base on the given frequency and the current `duty_resolution`.

Parameters

- `speed_mode`: Select the LEDC `speed_mode`, high-speed mode and low-speed mode
- `timer_num`: LEDC timer index (0-3), select from `ledc_timer_t`
- `freq_hz`: Set the LEDC frequency

`uint32_t ledc_get_freq` (*ledc_mode_t speed_mode, ledc_timer_t timer_num*)

LEDC get channel frequency (Hz)

Return

- 0 error
- Others Current LEDC frequency

Parameters

- `speed_mode`: Select the LEDC `speed_mode`, high-speed mode and low-speed mode
- `timer_num`: LEDC timer index (0-3), select from `ledc_timer_t`

`esp_err_t ledc_set_duty_with_hpoint` (*ledc_mode_t speed_mode, ledc_channel_t channel, uint32_t duty, uint32_t hpoint*)

LEDC set duty and hpoint value Only after calling `ledc_update_duty` will the duty update.

Note `ledc_set_duty`, `ledc_set_duty_with_hpoint` and `ledc_update_duty` are not thread-safe, do not call these functions to control one LEDC channel in different tasks at the same time. A thread-safe version of API is `ledc_set_duty_and_update`

Note If a fade operation is running in progress on that channel, the driver would not allow it to be stopped. Other duty operations will have to wait until the fade operation has finished.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `speed_mode`: Select the LEDC `speed_mode`, high-speed mode and low-speed mode
- `channel`: LEDC channel (0-7), select from `ledc_channel_t`
- `duty`: Set the LEDC duty, the range of duty setting is `[0, (2**duty_resolution)]`
- `hpoint`: Set the LEDC hpoint value(max: `0xffff`)

`int ledc_get_hpoint` (*ledc_mode_t speed_mode, ledc_channel_t channel*)

LEDC get hpoint value, the counter value when the output is set high level.

Return

- `LEDC_ERR_VAL` if parameter error
- Others Current hpoint value of LEDC channel

Parameters

- `speed_mode`: Select the LEDC `speed_mode`, high-speed mode and low-speed mode
- `channel`: LEDC channel (0-7), select from `ledc_channel_t`

`esp_err_t ledc_set_duty` (*ledc_mode_t speed_mode, ledc_channel_t channel, uint32_t duty*)

LEDC set duty This function do not change the hpoint value of this channel. if needed, please call `ledc_set_duty_with_hpoint`. only after calling `ledc_update_duty` will the duty update.

Note `ledc_set_duty`, `ledc_set_duty_with_hpoint` and `ledc_update_duty` are not thread-safe, do not call these functions to control one LEDC channel in different tasks at the same time. A thread-safe version of API is `ledc_set_duty_and_update`.

Note If a fade operation is running in progress on that channel, the driver would not allow it to be stopped. Other duty operations will have to wait until the fade operation has finished.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `speed_mode`: Select the LEDC `speed_mode`, high-speed mode and low-speed mode
- `channel`: LEDC channel (0-7), select from `ledc_channel_t`
- `duty`: Set the LEDC duty, the range of duty setting is $[0, (2^{**}duty_resolution)]$

`uint32_t ledc_get_duty` (*`ledc_mode_t speed_mode, ledc_channel_t channel`*)

LEDC get duty This function returns the duty at the present PWM cycle. You shouldn't expect the function to return the new duty in the same cycle of calling `ledc_update_duty`, because duty update doesn't take effect until the next cycle.

Return

- LEDC_ERR_DUTY if parameter error
- Others Current LEDC duty

Parameters

- `speed_mode`: Select the LEDC `speed_mode`, high-speed mode and low-speed mode
- `channel`: LEDC channel (0-7), select from `ledc_channel_t`

`esp_err_t ledc_set_fade` (`ledc_mode_t speed_mode, ledc_channel_t channel, uint32_t duty, ledc_duty_direction_t fade_direction, uint32_t step_num, uint32_t duty_cycle_num, uint32_t duty_scale`)

LEDC set gradient Set LEDC gradient, After the function calls the `ledc_update_duty` function, the function can take effect.

Note If a fade operation is running in progress on that channel, the driver would not allow it to be stopped. Other duty operations will have to wait until the fade operation has finished.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `speed_mode`: Select the LEDC `speed_mode`, high-speed mode and low-speed mode
- `channel`: LEDC channel (0-7), select from `ledc_channel_t`
- `duty`: Set the start of the gradient duty, the range of duty setting is $[0, (2^{**}duty_resolution)]$
- `fade_direction`: Set the direction of the gradient
- `step_num`: Set the number of the gradient
- `duty_cycle_num`: Set how many LEDC tick each time the gradient lasts
- `duty_scale`: Set gradient change amplitude

*`esp_err_t ledc_isr_register` (`void (*fn)`) `void *`*

, `void *arg`, `int intr_alloc_flags`, *`ledc_isr_handle_t *handle`* Register LEDC interrupt handler, the handler is an ISR. The handler will be attached to the same CPU core that this function is running on.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Function pointer error.

Parameters

- `fn`: Interrupt handler function.
- `arg`: User-supplied argument passed to the handler function.
- `intr_alloc_flags`: Flags used to allocate the interrupt. One or multiple (ORred) `ESP_INTR_FLAG_*` values. See `esp_intr_alloc.h` for more info.
- `handle`: Pointer to return handle. If non-NULL, a handle for the interrupt will be returned here.

`esp_err_t ledc_timer_set` (`ledc_mode_t speed_mode, ledc_timer_t timer_sel, uint32_t clock_divider, uint32_t duty_resolution, ledc_clk_src_t clk_src`)

Configure LEDC settings.

Return

- (-1) Parameter error
- Other Current LEDC duty

Parameters

- `speed_mode`: Select the LEDC `speed_mode`, high-speed mode and low-speed mode
- `timer_sel`: Timer index (0-3), there are 4 timers in LEDC module
- `clock_divider`: Timer clock divide value, the timer clock is divided from the selected clock source
- `duty_resolution`: Resolution of duty setting in number of bits. The range of duty values is [0, (2**`duty_resolution`)]
- `clk_src`: Select LEDC source clock.

esp_err_t **ledc_timer_rst** (*ledc_mode_t* `speed_mode`, *ledc_timer_t* `timer_sel`)

Reset LEDC timer.

Return

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

Parameters

- `speed_mode`: Select the LEDC `speed_mode`, high-speed mode and low-speed mode
- `timer_sel`: LEDC timer index (0-3), select from `ledc_timer_t`

esp_err_t **ledc_timer_pause** (*ledc_mode_t* `speed_mode`, *ledc_timer_t* `timer_sel`)

Pause LEDC timer counter.

Return

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

Parameters

- `speed_mode`: Select the LEDC `speed_mode`, high-speed mode and low-speed mode
- `timer_sel`: LEDC timer index (0-3), select from `ledc_timer_t`

esp_err_t **ledc_timer_resume** (*ledc_mode_t* `speed_mode`, *ledc_timer_t* `timer_sel`)

Resume LEDC timer.

Return

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

Parameters

- `speed_mode`: Select the LEDC `speed_mode`, high-speed mode and low-speed mode
- `timer_sel`: LEDC timer index (0-3), select from `ledc_timer_t`

esp_err_t **ledc_bind_channel_timer** (*ledc_mode_t* `speed_mode`, *ledc_channel_t* `channel`, *ledc_timer_t* `timer_sel`)

Bind LEDC channel with the selected timer.

Return

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

Parameters

- `speed_mode`: Select the LEDC `speed_mode`, high-speed mode and low-speed mode
- `channel`: LEDC channel index (0-7), select from `ledc_channel_t`
- `timer_sel`: LEDC timer index (0-3), select from `ledc_timer_t`

esp_err_t **ledc_set_fade_with_step** (*ledc_mode_t* `speed_mode`, *ledc_channel_t* `channel`, *uint32_t* `target_duty`, *uint32_t* `scale`, *uint32_t* `cycle_num`)

Set LEDC fade function.

Note Call `ledc_fade_func_install()` once before calling this function. Call `ledc_fade_start()` after this to start fading.

Note `ledc_set_fade_with_step`, `ledc_set_fade_with_time` and `ledc_fade_start` are not thread-safe, do not call these functions to control one LEDC channel in different tasks at the same time. A thread-safe version of API is `ledc_set_fade_step_and_start`

Note If a fade operation is running in progress on that channel, the driver would not allow it to be stopped. Other duty operations will have to wait until the fade operation has finished.

Return

- `ESP_ERR_INVALID_ARG` Parameter error

- ESP_OK Success
- ESP_ERR_INVALID_STATE Fade function not installed.
- ESP_FAIL Fade function init error

Parameters

- `speed_mode`: Select the LEDC `speed_mode`, high-speed mode and low-speed mode,
- `channel`: LEDC channel index (0-7), select from `ledc_channel_t`
- `target_duty`: Target duty of fading $[0, (2^{**}duty_resolution) - 1]$
- `scale`: Controls the increase or decrease step scale.
- `cycle_num`: increase or decrease the duty every `cycle_num` cycles

esp_err_t **ledc_set_fade_with_time** (*ledc_mode_t* `speed_mode`, *ledc_channel_t* `channel`, *uint32_t* `target_duty`, *int* `max_fade_time_ms`)

Set LEDC fade function, with a limited time.

Note Call `ledc_fade_func_install()` once before calling this function. Call `ledc_fade_start()` after this to start fading.

Note `ledc_set_fade_with_step`, `ledc_set_fade_with_time` and `ledc_fade_start` are not thread-safe, do not call these functions to control one LEDC channel in different tasks at the same time. A thread-safe version of API is `ledc_set_fade_step_and_start`

Note If a fade operation is running in progress on that channel, the driver would not allow it to be stopped. Other duty operations will have to wait until the fade operation has finished.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success
- ESP_ERR_INVALID_STATE Fade function not installed.
- ESP_FAIL Fade function init error

Parameters

- `speed_mode`: Select the LEDC `speed_mode`, high-speed mode and low-speed mode,
- `channel`: LEDC channel index (0-7), select from `ledc_channel_t`
- `target_duty`: Target duty of fading. $(0 - (2^{**} duty_resolution - 1))$
- `max_fade_time_ms`: The maximum time of the fading (ms).

esp_err_t **ledc_fade_func_install** (*int* `intr_alloc_flags`)

Install LEDC fade function. This function will occupy interrupt of LEDC module.

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE Fade function already installed.

Parameters

- `intr_alloc_flags`: Flags used to allocate the interrupt. One or multiple (ORred) `ESP_INTR_FLAG_*` values. See `esp_intr_alloc.h` for more info.

void **ledc_fade_func_uninstall** (*void*)

Uninstall LEDC fade function.

esp_err_t **ledc_fade_start** (*ledc_mode_t* `speed_mode`, *ledc_channel_t* `channel`, *ledc_fade_mode_t* `fade_mode`)

Start LEDC fading.

Note Call `ledc_fade_func_install()` once before calling this function. Call this API right after `ledc_set_fade_with_time` or `ledc_set_fade_with_step` before to start fading.

Note If a fade operation is running in progress on that channel, the driver would not allow it to be stopped. Other duty operations will have to wait until the fade operation has finished.

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE Fade function not installed.
- ESP_ERR_INVALID_ARG Parameter error.

Parameters

- `speed_mode`: Select the LEDC `speed_mode`, high-speed mode and low-speed mode
- `channel`: LEDC channel number
- `fade_mode`: Whether to block until fading done. See `ledc_types.h` `ledc_fade_mode_t` for more info. Note that this function will not return until fading to the target duty if

LEDC_FADE_WAIT_DONE mode is selected.

esp_err_t **ledc_set_duty_and_update** (*ledc_mode_t* speed_mode, *ledc_channel_t* channel, uint32_t duty, uint32_t hpoint)

A thread-safe API to set duty for LEDC channel and return when duty updated.

Note If a fade operation is running in progress on that channel, the driver would not allow it to be stopped. Other duty operations will have to wait until the fade operation has finished.

Parameters

- speed_mode: Select the LEDC speed_mode, high-speed mode and low-speed mode
- channel: LEDC channel (0-7), select from ledc_channel_t
- duty: Set the LEDC duty, the range of duty setting is [0, (2**duty_resolution)]
- hpoint: Set the LEDC hpoint value(max: 0xffff)

esp_err_t **ledc_set_fade_time_and_start** (*ledc_mode_t* speed_mode, *ledc_channel_t* channel, uint32_t target_duty, uint32_t max_fade_time_ms, *ledc_fade_mode_t* fade_mode)

A thread-safe API to set and start LEDC fade function, with a limited time.

Note Call ledc_fade_func_install() once, before calling this function.

Note If a fade operation is running in progress on that channel, the driver would not allow it to be stopped. Other duty operations will have to wait until the fade operation has finished.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success
- ESP_ERR_INVALID_STATE Fade function not installed.
- ESP_FAIL Fade function init error

Parameters

- speed_mode: Select the LEDC speed_mode, high-speed mode and low-speed mode,
- channel: LEDC channel index (0-7), select from ledc_channel_t
- target_duty: Target duty of fading.(0 - (2 ** duty_resolution - 1))
- max_fade_time_ms: The maximum time of the fading (ms).
- fade_mode: choose blocking or non-blocking mode

esp_err_t **ledc_set_fade_step_and_start** (*ledc_mode_t* speed_mode, *ledc_channel_t* channel, uint32_t target_duty, uint32_t scale, uint32_t cycle_num, *ledc_fade_mode_t* fade_mode)

A thread-safe API to set and start LEDC fade function.

Note Call ledc_fade_func_install() once before calling this function.

Note If a fade operation is running in progress on that channel, the driver would not allow it to be stopped. Other duty operations will have to wait until the fade operation has finished.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success
- ESP_ERR_INVALID_STATE Fade function not installed.
- ESP_FAIL Fade function init error

Parameters

- speed_mode: Select the LEDC speed_mode, high-speed mode and low-speed mode,
- channel: LEDC channel index (0-7), select from ledc_channel_t
- target_duty: Target duty of fading [0, (2**duty_resolution) - 1]
- scale: Controls the increase or decrease step scale.
- cycle_num: increase or decrease the duty every cycle_num cycles
- fade_mode: choose blocking or non-blocking mode

Macros

LEDC_APB_CLK_HZ

LEDC_REF_CLK_HZ

LEDC_ERR_DUTY

LEDC_ERR_VAL

Type Definitions

typedef *intr_handle_t* **ledc_isr_handle_t**

Header File

- [soc/include/hal/ledc_types.h](#)

Structures

struct **ledc_channel_config_t**

Configuration parameters of LEDC channel for `ledc_channel_config` function.

Public Members

int **gpio_num**

the LEDC output `gpio_num`, if you want to use `gpio16`, `gpio_num = 16`

ledc_mode_t **speed_mode**

LEDC speed `speed_mode`, high-speed mode or low-speed mode

ledc_channel_t **channel**

LEDC channel (0 - 7)

ledc_intr_type_t **intr_type**

configure interrupt, Fade interrupt enable or Fade interrupt disable

ledc_timer_t **timer_sel**

Select the timer source of channel (0 - 3)

uint32_t **duty**

LEDC channel duty, the range of duty setting is $[0, (2^{**}duty_resolution)]$

int **hpoint**

LEDC channel `hpoint` value, the max value is `0xffff`

struct **ledc_timer_config_t**

Configuration parameters of LEDC Timer timer for `ledc_timer_config` function.

Public Members

ledc_mode_t **speed_mode**

LEDC speed `speed_mode`, high-speed mode or low-speed mode

ledc_timer_bit_t **duty_resolution**

LEDC channel duty resolution

ledc_timer_bit_t **bit_num**

Deprecated in ESP-IDF 3.0. This is an alias to ‘`duty_resolution`’ for backward compatibility with ESP-IDF 2.1

ledc_timer_t **timer_num**

The timer source of channel (0 - 3)

uint32_t **freq_hz**

LEDC timer frequency (Hz)

ledc_clk_cfg_t **clk_cfg**

Configure LEDC source clock. For low speed channels and high speed channels, you can specify the source clock using `LEDC_USE_REF_TICK`, `LEDC_USE_APB_CLK` or `LEDC_AUTO_CLK`. For low speed channels, you can also specify the source clock using `LEDC_USE_RTC8M_CLK`, in this case, all low speed channel’s source clock must be `RTC8M_CLK`

Enumerations**enum ledc_mode_t***Values:***LEDC_LOW_SPEED_MODE**
LEDC low speed speed_mode**LEDC_SPEED_MODE_MAX**
LEDC speed limit**enum ledc_intr_type_t***Values:***LEDC_INTR_DISABLE = 0**
Disable LEDC interrupt**LEDC_INTR_FADE_END**
Enable LEDC interrupt**LEDC_INTR_MAX****enum ledc_duty_direction_t***Values:***LEDC_DUTY_DIR_DECREASE = 0**
LEDC duty decrease direction**LEDC_DUTY_DIR_INCREASE = 1**
LEDC duty increase direction**LEDC_DUTY_DIR_MAX****enum ledc_slow_clk_sel_t***Values:***LEDC_SLOW_CLK_RTC8M = 0**
LEDC low speed timer clock source is 8MHz RTC clock**LEDC_SLOW_CLK_APB**
LEDC low speed timer clock source is 80MHz APB clock**enum ledc_clk_cfg_t***Values:***LEDC_AUTO_CLK = 0**
The driver will automatically select the source clock(REF_TICK or APB) based on the giving resolution and duty parameter when init the timer**LEDC_USE_REF_TICK**
LEDC timer select REF_TICK clock as source clock**LEDC_USE_APB_CLK**
LEDC timer select APB clock as source clock**LEDC_USE_RTC8M_CLK**
LEDC timer select RTC8M_CLK as source clock. Only for low speed channels and this parameter must be the same for all low speed channels**enum ledc_clk_src_t***Values:***LEDC_REF_TICK = LEDC_USE_REF_TICK**
LEDC timer clock divided from reference tick (1Mhz)**LEDC_APB_CLK = LEDC_USE_APB_CLK**
LEDC timer clock divided from APB clock (80Mhz)**enum ledc_timer_t***Values:*

LEDC_TIMER_0 = 0
LEDC timer 0

LEDC_TIMER_1
LEDC timer 1

LEDC_TIMER_2
LEDC timer 2

LEDC_TIMER_3
LEDC timer 3

LEDC_TIMER_MAX

enum ledc_channel_t

Values:

LEDC_CHANNEL_0 = 0
LEDC channel 0

LEDC_CHANNEL_1
LEDC channel 1

LEDC_CHANNEL_2
LEDC channel 2

LEDC_CHANNEL_3
LEDC channel 3

LEDC_CHANNEL_4
LEDC channel 4

LEDC_CHANNEL_5
LEDC channel 5

LEDC_CHANNEL_6
LEDC channel 6

LEDC_CHANNEL_7
LEDC channel 7

LEDC_CHANNEL_MAX

enum ledc_timer_bit_t

Values:

LEDC_TIMER_1_BIT = 1
LEDC PWM duty resolution of 1 bits

LEDC_TIMER_2_BIT
LEDC PWM duty resolution of 2 bits

LEDC_TIMER_3_BIT
LEDC PWM duty resolution of 3 bits

LEDC_TIMER_4_BIT
LEDC PWM duty resolution of 4 bits

LEDC_TIMER_5_BIT
LEDC PWM duty resolution of 5 bits

LEDC_TIMER_6_BIT
LEDC PWM duty resolution of 6 bits

LEDC_TIMER_7_BIT
LEDC PWM duty resolution of 7 bits

LEDC_TIMER_8_BIT
LEDC PWM duty resolution of 8 bits

LEDC_TIMER_9_BIT
LEDC PWM duty resolution of 9 bits

LEDC_TIMER_10_BIT
LEDC PWM duty resolution of 10 bits

LEDC_TIMER_11_BIT
LEDC PWM duty resolution of 11 bits

LEDC_TIMER_12_BIT
LEDC PWM duty resolution of 12 bits

LEDC_TIMER_13_BIT
LEDC PWM duty resolution of 13 bits

LEDC_TIMER_14_BIT
LEDC PWM duty resolution of 14 bits

LEDC_TIMER_15_BIT
LEDC PWM duty resolution of 15 bits

LEDC_TIMER_16_BIT
LEDC PWM duty resolution of 16 bits

LEDC_TIMER_17_BIT
LEDC PWM duty resolution of 17 bits

LEDC_TIMER_18_BIT
LEDC PWM duty resolution of 18 bits

LEDC_TIMER_19_BIT
LEDC PWM duty resolution of 19 bits

LEDC_TIMER_20_BIT
LEDC PWM duty resolution of 20 bits

LEDC_TIMER_BIT_MAX

enum ledc_fade_mode_t

Values:

LEDC_FADE_NO_WAIT = 0
LEDC fade function will return immediately

LEDC_FADE_WAIT_DONE
LEDC fade function will block until fading to the target duty

LEDC_FADE_MAX

2.3.8 MCPWM

ESP32 has two MCPWM units which can be used to control different types of motors. Each unit has three pairs of PWM outputs.

Further in documentation the outputs of a single unit are labeled `PWMxA` / `PWMxB`.

More detailed block diagram of the MCPWM unit is shown below. Each A/B pair may be clocked by any one of the three timers Timer 0, 1 and 2. The same timer may be used to clock more than one pair of PWM outputs. Each unit is also able to collect inputs such as `SYNC SIGNALS`, detect `FAULT SIGNALS` like motor overcurrent or overvoltage, as well as obtain feedback with `CAPTURE SIGNALS` on e.g. a rotor position.

Description of this API starts with configuration of MCPWM's **Timer** and **Generator** submodules to provide the basic motor control functionality. Then it discusses more advanced submodules and functionalities of a **Fault Handler**, signal **Capture**, **Carrier** and **Interrupts**.

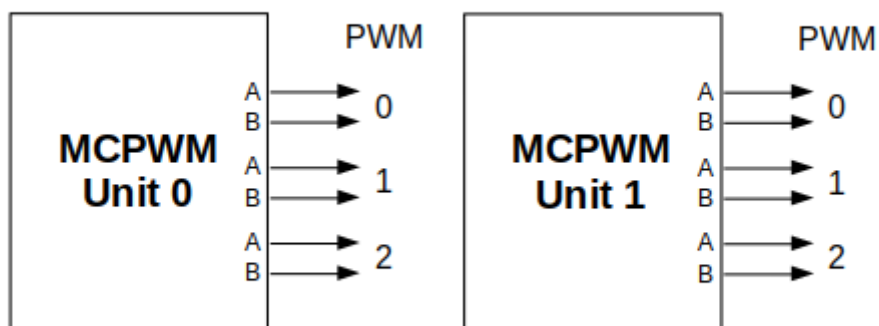


Fig. 15: MCPWM Overview

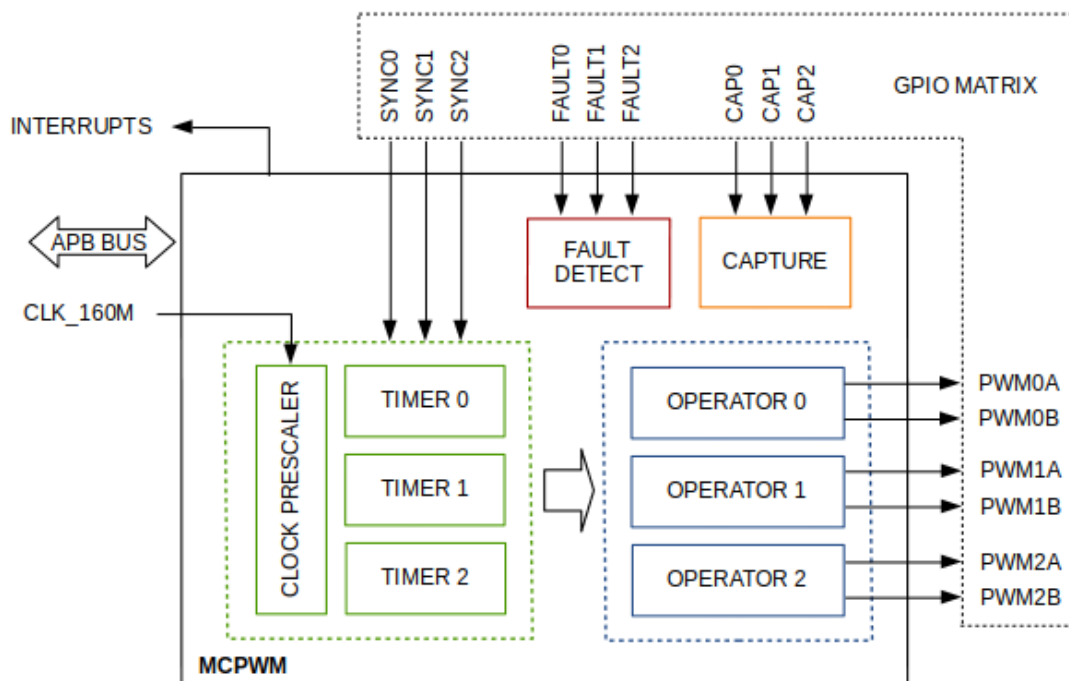


Fig. 16: MCPWM Block Diagram

Contents

- *Configure* a basic functionality of the outputs
- *Operate* the outputs to drive a motor
- *Adjust* how the motor is driven
- *Capture* external signals to provide additional control over the outputs
- Use *Fault Handler* to detect and manage faults
- Add a higher frequency *Carrier*, if output signals are passed through an isolation transformer
- Configuration and handling of *Interrupts*.

Configure

The scope of configuration depends on the motor type, in particular how many outputs and inputs are required, and what will be the sequence of signals to drive the motor.

In this case we will describe a simple configuration to control a brushed DC motor that is using only some of the available MCPWM's resources. An example circuit is shown below. It includes a [H-Bridge](#) to switch polarization of a voltage applied to the motor (M) and to provide sufficient current to drive it.

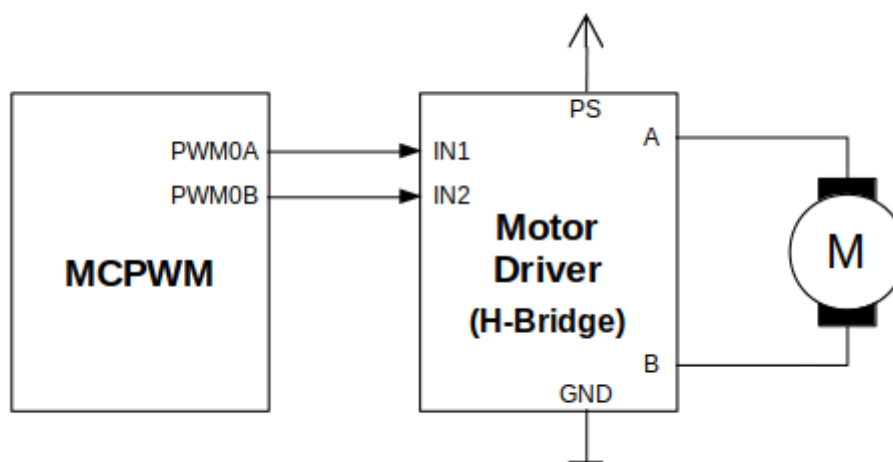


Fig. 17: Example of Brushed DC Motor Control with MCPWM

Configuration covers the following steps:

1. Selection of a MPWn unit that will be used to drive the motor. There are two units available on-board of ESP32 and enumerated in `mcpwm_unit_t`.
2. Initialization of two GPIOs as output signals within selected unit by calling `mcpwm_gpio_init()`. The two output signals are typically used to command the motor to rotate right or left. All available signal options are listed in `mcpwm_io_signals_t`. To set more than a single pin at a time, use function `mcpwm_set_pin()` together with `mcpwm_pin_config_t`.
3. Selection of a timer. There are three timers available within the unit. The timers are listed in `mcpwm_timer_t`.
4. Setting of the timer frequency and initial duty within `mcpwm_config_t` structure.
5. Calling of `mcpwm_init()` with the above parameters to make the configuration effective.

Operate

To operate a motor connected to the MCPWM unit, e.g. turn it left or right, or vary the speed, we should apply some control signals to the unit's outputs. The outputs are organized into three pairs. Within a pair they are labeled "A" and "B" and each driven by a submodule called an "Generator". To provide a PWM signal, the Operator itself,

which contains two Generator, should be clocked by one of three available Timers. To make the API simpler, each Timer is automatically associated by the API to drive an Operator of the same index, e.g. Timer 0 is associated with Operator 0.

There are the following basic ways to control the outputs:

- We can drive particular signal steady high or steady low with function `mcpwm_set_signal_high()` or `mcpwm_set_signal_low()`. This will make the motor to turn with a maximum speed or stop. Depending on selected output A or B the motor will rotate either right or left.
- Another option is to drive the outputs with the PWM signal by calling `mcpwm_start()` or `mcpwm_stop()`. The motor speed will be proportional to the PWM duty.
- To vary PWM's duty call `mcpwm_set_duty()` and provide the duty value in %. Optionally, you may call `mcpwm_set_duty_in_us()`, if you prefer to set the duty in microseconds. Checking of currently set value is possible by calling `mcpwm_get_duty()`. Phase of the PWM signal may be altered by calling `mcpwm_set_duty_type()`. The duty is set individually for each A and B output using `mcpwm_generator_t` in specific function calls. The duty value refers either to high or low output signal duration. This is configured when calling `mcpwm_init()`, as discussed in section [Configure](#), and selecting one of options from `mcpwm_duty_type_t`.

Note: Call function `mcpwm_set_duty_type()` every time after `mcpwm_set_signal_high()` or `mcpwm_set_signal_low()` to resume with previously set duty cycle.

Adjust

There are couple of ways to adjust a signal on the outputs and changing how the motor operates.

- Set specific PWM frequency by calling `mcpwm_set_frequency()`. This may be required to adjust to electrical or mechanical characteristics of particular motor and driver. To check what frequency is set, use function `mcpwm_get_frequency()`.
- Introduce a dead time between outputs A and B when they are changing the state to reverse direction of the motor rotation. This is to make up for on/off switching delay of the motor driver FETs. The dead time options are defined in `mcpwm_deadtime_type_t` and enabled by calling `mcpwm_deadtime_enable()`. To disable this functionality call `mcpwm_deadtime_disable()`.
- Synchronize outputs of operator submodules, e.g. to get raising edge of PWM0A/B and PWM1A/B to start exactly at the same time, or shift them between each other by a given phase. Synchronization is triggered by SYNC SIGNALS shown on the [block diagram](#) of the MCPWM above, and defined in `mcpwm_sync_signal_t`. To attach the signal to a GPIO call `mcpwm_gpio_init()`. You can then enable synchronization with function `mcpwm_sync_enable()`. As input parameters provide MCPWM unit, timer to synchronize, the synchronization signal and a phase to delay the timer.

Note: Synchronization signals are referred to using two different enumerations. First one `mcpwm_io_signals_t` is used together with function `mcpwm_gpio_init()` when selecting a GPIO as the signal input source. The second one `mcpwm_sync_signal_t` is used when enabling or disabling synchronization with `mcpwm_sync_enable()` or `mcpwm_sync_disable()`.

- Vary the pattern of the A/B output signals by getting MCPWM counters to count up, down and up/down (automatically changing the count direction). Respective configuration is done when calling `mcpwm_init()`, as discussed in section [Configure](#), and selecting one of counter types from `mcpwm_counter_type_t`. For explanation of how A/B PWM output signals are generated please refer to [ESP32 Technical Reference Manual](#).

Capture

One of requirements of BLDC (Brushless DC, see figure below) motor control is sensing of the rotor position. To facilitate this task each MCPWM unit provides three sensing inputs together with dedicated hardware. The hardware is able to detect the input signal's edge and measure time between signals. As result the control software is simpler and the CPU power may be used for other tasks.

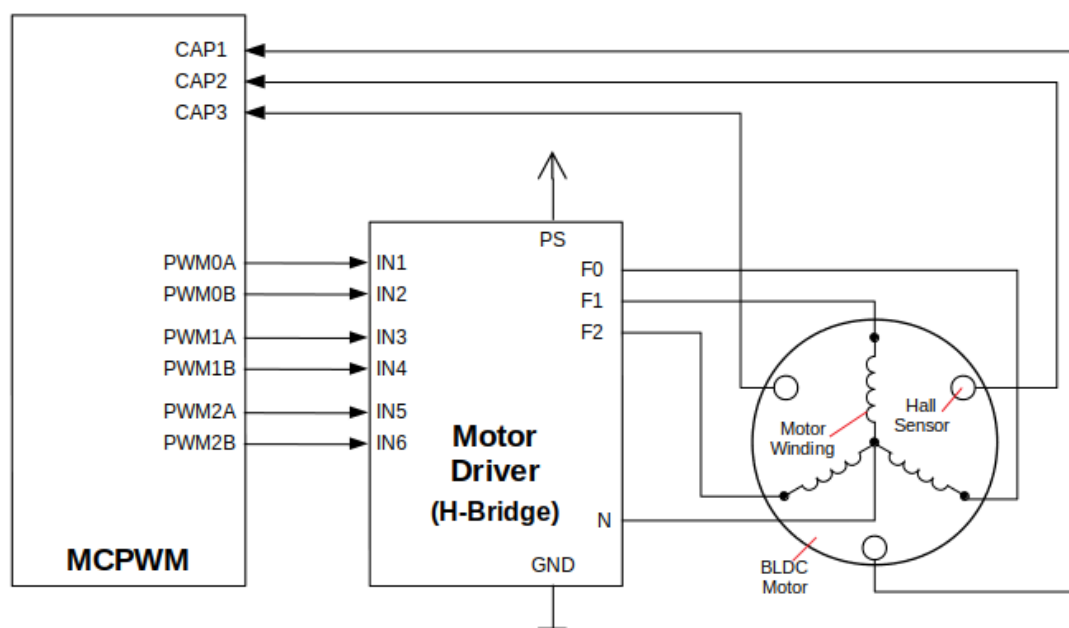


Fig. 18: Example of Brushless DC Motor Control with MCPWM

The capture functionality may be used for other types of motors or tasks. The functionality is enabled in two steps:

1. Configuration of GPIOs to act as the capture signal inputs by calling functions `mcpwm_gpio_init()` or `mcpwm_set_pin()`, that were described in section [Configure](#).
2. Enabling of the functionality itself by invoking `mcpwm_capture_enable()`, selecting desired signal input from `mcpwm_capture_signal_t`, setting the signal edge with `mcpwm_capture_on_edge_t` and the signal count prescaler.

Within the second step above a 32-bit capture timer is enabled. The timer runs continuously driven by the APB clock. The clock frequency is typically 80 MHz. On each capture event the capture timer's value is stored in time-stamp register that may be then checked by calling `mcpwm_capture_signal_get_value()`. The edge of the last signal may be checked with `mcpwm_capture_signal_get_edge()`.

If not required anymore, the capture functionality may be disabled with `mcpwm_capture_disable()`.

Fault Handler

Each unit of the MCPWM is able to sense external signals with information about failure of the motor, the motor driver or any other device connected to the MCPWM. There are three fault inputs per unit that may be routed to user selectable GPIOs. The MCPWM may be configured to perform one of four predefined actions on A/B outputs when a fault signal is received:

- lock current state of the output
- set the output low
- set the output high
- toggle the output

The user should determine possible failure modes of the motor and what action should be performed on detection of particular fault, e.g. drive all outputs low for a brushed motor, or lock current state for a stepper motor, etc. As result of this action the motor should be put into a safe state to reduce likelihood of a damage caused by the fault.

The fault handler functionality is enabled in two steps:

1. Configuration of GPIOs to act as fault signal inputs. This is done in analogous way as described for capture signals in section above. It includes setting the signal level to trigger the fault as defined in [mcpwm_fault_input_level_t](#).
2. Initialization of the fault handler by calling either [mcpwm_fault_set_oneshot_mode\(\)](#) or [mcpwm_fault_set_cyc_mode\(\)](#). These functions set the mode that MCPWM should operate once fault signal becomes inactive. There are two modes possible:
 - State of MCPWM unit will be locked until reset - [mcpwm_fault_set_oneshot_mode\(\)](#).
 - The MCPWM will resume operation once fault signal becoming inactive - [mcpwm_fault_set_cyc_mode\(\)](#).

The function call parameters include selection of one of three fault inputs defined in [mcpwm_fault_signal_t](#) and specific action on outputs A and B defined in [mcpwm_action_on_pwmxa_t](#) and [mcpwm_action_on_pwmxb_t](#).

Particular fault signal may be disabled at the runtime by calling [mcpwm_fault_deinit\(\)](#).

Carrier

The MCPWM has a carrier submodule used if galvanic isolation from the motor driver is required by passing the A/B output signals through transformers. Any of A and B output signals may be at 100% duty and not changing whenever motor is required to run steady at the full load. Coupling of non alternating signals with a transformer is problematic, so the signals are modulated by the carrier submodule to create an AC waveform, to make the coupling possible.

To use the carrier submodule, it should be first initialized by calling [mcpwm_carrier_init\(\)](#). The carrier parameters are defined in [mcpwm_carrier_config_t](#) structure invoked within the function call. Then the carrier functionality may be enabled by calling [mcpwm_carrier_enable\(\)](#).

The carrier parameters may be then alerted at a runtime by calling dedicated functions to change individual fields of the [mcpwm_carrier_config_t](#) structure, like [mcpwm_carrier_set_period\(\)](#), [mcpwm_carrier_set_duty_cycle\(\)](#), [mcpwm_carrier_output_invert\(\)](#), etc.

This includes enabling and setting duration of the first pulse of the career with [mcpwm_carrier_oneshot_mode_enable\(\)](#). For more details please refer to “PWM Carrier Submodule” section of the [ESP32 Technical Reference Manual](#).

To disable carrier functionality call [mcpwm_carrier_disable\(\)](#).

Interrupts

Registering of the MCPWM interrupt handler is possible by calling [mcpwm_isr_register\(\)](#).

Application Example

Examples of using MCPWM for motor control: [peripherals/mcpwm](#):

- Demonstration how to use each submodule of the MCPWM - [peripherals/mcpwm/mcpwm_basic_config](#)
- Control of BLDC (brushless DC) motor with hall sensor feedback - [peripherals/mcpwm/mcpwm_bldc_control](#)
- Brushed DC motor control - [peripherals/mcpwm/mcpwm_brushed_dc_control](#)
- Servo motor control - [peripherals/mcpwm/mcpwm_servo_control](#)

API Reference

Header File

- [soc/include/hal/mcpwm_types.h](#)

Enumerations**enum mcpwm_intr_t**

Interrupts for MCPWM.

*Values:***MCPWM_LL_INTR_CAP0** = BIT(27)

Capture 0 happened.

MCPWM_LL_INTR_CAP1 = BIT(28)

Capture 1 happened.

MCPWM_LL_INTR_CAP2 = BIT(29)

Capture 2 happened.

enum mcpwm_counter_type_t

Select type of MCPWM counter.

*Values:***MCPWM_UP_COUNTER** = 1

For asymmetric MCPWM

MCPWM_DOWN_COUNTER

For asymmetric MCPWM

MCPWM_UP_DOWN_COUNTER

For symmetric MCPWM, frequency is half of MCPWM frequency set

MCPWM_COUNTER_MAX

Maximum counter mode

enum mcpwm_duty_type_t

Select type of MCPWM duty cycle mode.

*Values:***MCPWM_DUTY_MODE_0** = 0

Active high duty, i.e. duty cycle proportional to high time for asymmetric MCPWM

MCPWM_DUTY_MODE_1

Active low duty, i.e. duty cycle proportional to low time for asymmetric MCPWM, out of phase(inverted) MCPWM

MCPWM_HAL_GENERATOR_MODE_FORCE_LOW**MCPWM_HAL_GENERATOR_MODE_FORCE_HIGH****MCPWM_DUTY_MODE_MAX**

Num of duty cycle modes

enum mcpwm_output_action_t

MCPWM select action to be taken on the output when event happens.

*Values:***MCPWM_ACTION_NO_CHANGE** = 0

No change in the output

MCPWM_ACTION_FORCE_LOW

Make output low

MCPWM_ACTION_FORCE_HIGH

Make output high

MCPWM_ACTION_TOGGLE

Make output toggle

enum mcpwm_deadtime_type_t

MCPWM deadtime types, used to generate deadtime, RED refers to rising edge delay and FED refers to falling edge delay.

Values:

MCPWM_DEADTIME_BYPASS = 0

Bypass the deadtime

MCPWM_BYPASS_RED

MCPWMXA Out = MCPWMXA In with no delay, MCPWMXB Out = MCPWMXA In with falling edge delay

MCPWM_BYPASS_FED

MCPWMXA Out = MCPWMXA In with rising edge delay, MCPWMXB Out = MCPWMXB In with no delay

MCPWM_ACTIVE_HIGH_MODE

MCPWMXA Out = MCPWMXA In with rising edge delay, MCPWMXB Out = MCPWMXA In with falling edge delay

MCPWM_ACTIVE_LOW_MODE

MCPWMXA Out = MCPWMXA In with compliment of rising edge delay, MCPWMXB Out = MCPWMXA In with compliment of falling edge delay

MCPWM_ACTIVE_HIGH_COMPLIMENT_MODE

MCPWMXA Out = MCPWMXA In with rising edge delay, MCPWMXB = MCPWMXA In with compliment of falling edge delay

MCPWM_ACTIVE_LOW_COMPLIMENT_MODE

MCPWMXA Out = MCPWMXA In with compliment of rising edge delay, MCPWMXB Out = MCPWMXA In with falling edge delay

MCPWM_ACTIVE_RED_FED_FROM_PWMXA

MCPWMXA Out = MCPWMXB Out = MCPWMXA In with rising edge delay as well as falling edge delay

MCPWM_ACTIVE_RED_FED_FROM_PWMXB

MCPWMXA Out = MCPWMXB Out = MCPWMXB In with rising edge delay as well as falling edge delay

MCPWM_DEADTIME_TYPE_MAX

enum mcpwm_sync_signal_t

MCPWM select sync signal input.

Values:

MCPWM_SELECT_SYNC0 = 4

Select SYNC0 as input

MCPWM_SELECT_SYNC1

Select SYNC1 as input

MCPWM_SELECT_SYNC2

Select SYNC2 as input

enum mcpwm_capture_on_edge_t

MCPWM select capture starts from which edge.

Values:

MCPWM_NEG_EDGE = BIT(0)

Capture the negative edge

MCPWM_POS_EDGE = BIT(1)

Capture the positive edge

Header File

- [driver/include/driver/mcpwm.h](#)

Functions

esp_err_t **mcpwm_gpio_init** (*mcpwm_unit_t* mcpwm_num, *mcpwm_io_signals_t* io_signal, int gpio_num)

This function initializes each gpio signal for MCPWM.

Note This function initializes one gpio at a time.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- mcpwm_num: set MCPWM unit(0-1)
- io_signal: set MCPWM signals, each MCPWM unit has 6 output(MCPWMXA, MCPWMXB) and 9 input(SYNC_X, FAULT_X, CAP_X) ‘X’ is timer_num(0-2)
- gpio_num: set this to configure gpio for MCPWM, if you want to use gpio16, gpio_num = 16

esp_err_t **mcpwm_set_pin** (*mcpwm_unit_t* mcpwm_num, **const** *mcpwm_pin_config_t* *mcpwm_pin)

Initialize MCPWM gpio structure.

Note This function can be used to initialize more then one gpio at a time.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- mcpwm_num: set MCPWM unit(0-1)
- mcpwm_pin: MCPWM pin structure

esp_err_t **mcpwm_init** (*mcpwm_unit_t* mcpwm_num, *mcpwm_timer_t* timer_num, **const** *mcpwm_config_t* *mcpwm_conf)

Initialize MCPWM parameters.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- mcpwm_num: set MCPWM unit(0-1)
- timer_num: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers.
- mcpwm_conf: configure structure *mcpwm_config_t*

esp_err_t **mcpwm_set_frequency** (*mcpwm_unit_t* mcpwm_num, *mcpwm_timer_t* timer_num, uint32_t frequency)

Set frequency(in Hz) of MCPWM timer.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- mcpwm_num: set MCPWM unit(0-1)
- timer_num: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- frequency: set the frequency in Hz of each timer

esp_err_t **mcpwm_set_duty** (*mcpwm_unit_t* mcpwm_num, *mcpwm_timer_t* timer_num, *mcpwm_generator_t* gen, float duty)

Set duty cycle of each operator(MCPWMXA/MCPWMXB)

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- mcpwm_num: set MCPWM unit(0-1)
- timer_num: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- gen: set the generator(MCPWMXA/MCPWMXB), ‘X’ is operator number selected
- duty: set duty cycle in %(i.e for 62.3% duty cycle, duty = 62.3) of each operator

esp_err_t **mcpwm_set_duty_in_us** (*mcpwm_unit_t* mcpwm_num, *mcpwm_timer_t* timer_num, *mcpwm_generator_t* gen, *uint32_t* duty_in_us)

Set duty cycle of each operator(MCPWMXA/MCPWMXB) in us.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- mcpwm_num: set MCPWM unit(0-1)
- timer_num: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- gen: set the generator(MCPWMXA/MCPWMXB), 'x' is operator number selected
- duty_in_us: set duty value in microseconds of each operator

esp_err_t **mcpwm_set_duty_type** (*mcpwm_unit_t* mcpwm_num, *mcpwm_timer_t* timer_num, *mcpwm_generator_t* gen, *mcpwm_duty_type_t* duty_type)

Set duty either active high or active low(out of phase/inverted)

Note Call this function every time after mcpwm_set_signal_high or mcpwm_set_signal_low to resume with previously set duty cycle

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- mcpwm_num: set MCPWM unit(0-1)
- timer_num: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- gen: set the generator(MCPWMXA/MCPWMXB), 'x' is operator number selected
- duty_type: set active low or active high duty type

uint32_t **mcpwm_get_frequency** (*mcpwm_unit_t* mcpwm_num, *mcpwm_timer_t* timer_num)

Get frequency of timer.

Return

- frequency of timer

Parameters

- mcpwm_num: set MCPWM unit(0-1)
- timer_num: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers

float **mcpwm_get_duty** (*mcpwm_unit_t* mcpwm_num, *mcpwm_timer_t* timer_num, *mcpwm_operator_t* gen)

Get duty cycle of each operator.

Return

- duty cycle in % of each operator(56.7 means duty is 56.7%)

Parameters

- mcpwm_num: set MCPWM unit(0-1)
- timer_num: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- gen: set the generator(MCPWMXA/MCPWMXB), 'x' is operator number selected

esp_err_t **mcpwm_set_signal_high** (*mcpwm_unit_t* mcpwm_num, *mcpwm_timer_t* timer_num, *mcpwm_generator_t* gen)

Use this function to set MCPWM signal high.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- mcpwm_num: set MCPWM unit(0-1)
- timer_num: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- gen: set the operator(MCPWMXA/MCPWMXB), 'x' is timer number selected

esp_err_t **mcpwm_set_signal_low** (*mcpwm_unit_t* mcpwm_num, *mcpwm_timer_t* timer_num, *mcpwm_generator_t* gen)

Use this function to set MCPWM signal low.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- `gen`: set the operator(MCPWMXA/MCPWMB), 'x' is timer number selected

esp_err_t **mcpwm_start** (*mcpwm_unit_t* `mcpwm_num`, *mcpwm_timer_t* `timer_num`)

Start MCPWM signal on timer 'x' .

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers

esp_err_t **mcpwm_stop** (*mcpwm_unit_t* `mcpwm_num`, *mcpwm_timer_t* `timer_num`)

Start MCPWM signal on timer 'x' .

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers

esp_err_t **mcpwm_carrier_init** (*mcpwm_unit_t* `mcpwm_num`, *mcpwm_timer_t* `timer_num`, **const** *mcpwm_carrier_config_t* *`carrier_conf`)

Initialize carrier configuration.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- `carrier_conf`: configure structure *mcpwm_carrier_config_t*

esp_err_t **mcpwm_carrier_enable** (*mcpwm_unit_t* `mcpwm_num`, *mcpwm_timer_t* `timer_num`)

Enable MCPWM carrier submodule, for respective timer.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers

esp_err_t **mcpwm_carrier_disable** (*mcpwm_unit_t* `mcpwm_num`, *mcpwm_timer_t* `timer_num`)

Disable MCPWM carrier submodule, for respective timer.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers

esp_err_t **mcpwm_carrier_set_period** (*mcpwm_unit_t* `mcpwm_num`, *mcpwm_timer_t* `timer_num`, *uint8_t* `carrier_period`)

Set period of carrier.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- `carrier_period`: set the carrier period of each timer, carrier period = (carrier_period + 1)*800ns (carrier_period <= 15)

`esp_err_t mcpwm_carrier_set_duty_cycle` (*mcpwm_unit_t* mcpwm_num, *mcpwm_timer_t* timer_num, *uint8_t* carrier_duty)

Set duty_cycle of carrier.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- `carrier_duty`: set duty_cycle of carrier , carrier duty cycle = carrier_duty*12.5% (chop_duty <= 7)

`esp_err_t mcpwm_carrier_oneshot_mode_enable` (*mcpwm_unit_t* mcpwm_num, *mcpwm_timer_t* timer_num, *uint8_t* pulse_width)

Enable and set width of first pulse in carrier oneshot mode.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- `pulse_width`: set pulse width of first pulse in oneshot mode, width = (carrier period)*(pulse_width + 1) (pulse_width <= 15)

`esp_err_t mcpwm_carrier_oneshot_mode_disable` (*mcpwm_unit_t* mcpwm_num, *mcpwm_timer_t* timer_num)

Disable oneshot mode, width of first pulse = carrier period.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers

`esp_err_t mcpwm_carrier_output_invert` (*mcpwm_unit_t* mcpwm_num, *mcpwm_timer_t* timer_num, *mcpwm_carrier_out_ivt_t* carrier_ivt_mode)

Enable or disable carrier output inversion.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- `carrier_ivt_mode`: enable or disable carrier output inversion

`esp_err_t mcpwm_deadtime_enable` (*mcpwm_unit_t* mcpwm_num, *mcpwm_timer_t* timer_num, *mcpwm_deadtime_type_t* dt_mode, *uint32_t* red, *uint32_t* fed)

Enable and initialize deadtime for each MCPWM timer.

Return

- ESP_OK Success

- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- `dt_mode`: set deadtime mode
- `red`: set rising edge delay = `red*100ns`
- `fed`: set rising edge delay = `fed*100ns`

`esp_err_t mcpwm_deadtime_disable` (`mcpwm_unit_t` `mcpwm_num`, `mcpwm_timer_t` `timer_num`)

Disable deadtime on MCPWM timer.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers

`esp_err_t mcpwm_fault_init` (`mcpwm_unit_t` `mcpwm_num`, `mcpwm_fault_input_level_t` `input_level`,
`mcpwm_fault_signal_t` `fault_sig`)

Initialize fault submodule, currently low level triggering is not supported.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `input_level`: set fault signal level, which will cause fault to occur
- `fault_sig`: set the fault pin, which needs to be enabled

`esp_err_t mcpwm_fault_set_oneshot_mode` (`mcpwm_unit_t` `mcpwm_num`, `mcpwm_timer_t` `timer_num`,
`mcpwm_fault_signal_t` `fault_sig`,
`mcpwm_output_action_t` `action_on_pwmxa`,
`mcpwm_output_action_t` `action_on_pwmxb`)

Set oneshot mode on fault detection, once fault occur in oneshot mode reset is required to resume MCPWM signals.

Note currently low level triggering is not supported

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- `fault_sig`: set the fault pin, which needs to be enabled for oneshot mode
- `action_on_pwmxa`: action to be taken on MCPWMXA when fault occurs, either no change or high or low or toggle
- `action_on_pwmxb`: action to be taken on MCPWMXB when fault occurs, either no change or high or low or toggle

`esp_err_t mcpwm_fault_set_cyc_mode` (`mcpwm_unit_t` `mcpwm_num`, `mcpwm_timer_t` `timer_num`,
`mcpwm_fault_signal_t` `fault_sig`, `mcpwm_output_action_t` `action_on_pwmxa`,
`mcpwm_output_action_t` `action_on_pwmxb`)

Set cycle-by-cycle mode on fault detection, once fault occur in cyc mode MCPWM signal resumes as soon as fault signal becomes inactive.

Note currently low level triggering is not supported

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `mcpwm_num`: set MCPWM unit(0-1)

- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- `fault_sig`: set the fault pin, which needs to be enabled for cyc mode
- `action_on_pwmxa`: action to be taken on MCPWMXA when fault occurs, either no change or high or low or toggle
- `action_on_pwmxb`: action to be taken on MCPWMXB when fault occurs, either no change or high or low or toggle

`esp_err_t mcpwm_fault_deinit` (*mcpwm_unit_t* `mcpwm_num`, *mcpwm_fault_signal_t* `fault_sig`)

Disable fault signal.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `fault_sig`: fault pin, which needs to be disabled

`esp_err_t mcpwm_capture_enable` (*mcpwm_unit_t* `mcpwm_num`, *mcpwm_capture_signal_t* `cap_sig`, *mcpwm_capture_on_edge_t* `cap_edge`, *uint32_t* `num_of_pulse`)

Initialize capture submodule.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `cap_edge`: set capture edge, `BIT(0)` - negative edge, `BIT(1)` - positive edge
- `cap_sig`: capture pin, which needs to be enabled
- `num_of_pulse`: count time between rising/falling edge between 2 *(pulses mentioned), counter uses `APB_CLK`

`esp_err_t mcpwm_capture_disable` (*mcpwm_unit_t* `mcpwm_num`, *mcpwm_capture_signal_t* `cap_sig`)

Disable capture signal.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `cap_sig`: capture pin, which needs to be disabled

uint32_t `mcpwm_capture_signal_get_value` (*mcpwm_unit_t* `mcpwm_num`, *mcpwm_capture_signal_t* `cap_sig`)

Get capture value.

Return Captured value

Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `cap_sig`: capture pin on which value is to be measured

uint32_t `mcpwm_capture_signal_get_edge` (*mcpwm_unit_t* `mcpwm_num`, *mcpwm_capture_signal_t* `cap_sig`)

Get edge of capture signal.

Return Capture signal edge: 1 - positive edge, 2 - negative edge

Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `cap_sig`: capture pin of whose edge is to be determined

`esp_err_t mcpwm_sync_enable` (*mcpwm_unit_t* `mcpwm_num`, *mcpwm_timer_t* `timer_num`, *mcpwm_sync_signal_t* `sync_sig`, *uint32_t* `phase_val`)

Initialize sync submodule.

Return

- `ESP_OK` Success

- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- `sync_sig`: set the synchronization pin, which needs to be enabled
- `phase_val`: phase value in 1/1000 (for 86.7%, `phase_val = 867`) which timer moves to on sync signal

`esp_err_t mcpwm_sync_disable(mcpwm_unit_t mcpwm_num, mcpwm_timer_t timer_num)`

Disable sync submodule on given timer.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers

`esp_err_t mcpwm_isr_register(mcpwm_unit_t mcpwm_num, void (*fn)) void *`

, void *arg, int intr_alloc_flags, intr_handle_t *handle Register MCPWM interrupt handler, the handler is an ISR. the handler will be attached to the same CPU core that this function is running on.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Function pointer error.

Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `fn`: interrupt handler function.
- `arg`: user-supplied argument passed to the handler function.
- `intr_alloc_flags`: flags used to allocate the interrupt. One or multiple (ORred) ESP_INTR_FLAG_* values. see `esp_intr_alloc.h` for more info.
- `handle`: pointer to return handle. If non-NULL, a handle for the interrupt will be returned here.

Structures

`struct mcpwm_pin_config_t`

MCPWM pin number for.

Public Members

int `mcpwm0a_out_num`
MCPWM0A out pin

int `mcpwm0b_out_num`
MCPWM0A out pin

int `mcpwm1a_out_num`
MCPWM0A out pin

int `mcpwm1b_out_num`
MCPWM0A out pin

int `mcpwm2a_out_num`
MCPWM0A out pin

int `mcpwm2b_out_num`
MCPWM0A out pin

int `mcpwm_sync0_in_num`
SYNC0 in pin

int `mcpwm_sync1_in_num`
SYNC1 in pin


```
int mcpwm_sync2_in_num  
    SYNC2 in pin  
int mcpwm_fault0_in_num  
    FAULT0 in pin  
int mcpwm_fault1_in_num  
    FAULT1 in pin  
int mcpwm_fault2_in_num  
    FAULT2 in pin  
int mcpwm_cap0_in_num  
    CAP0 in pin  
int mcpwm_cap1_in_num  
    CAP1 in pin  
int mcpwm_cap2_in_num  
    CAP2 in pin  
struct mcpwm_config_t  
    MCPWM config structure.
```

Public Members

```
uint32_t frequency  
    Set frequency of MCPWM in Hz  
float cmp_r_a  
    Set % duty cycle for operator a(MCPWMXA), i.e for 62.3% duty cycle, duty_a = 62.3  
float cmp_r_b  
    Set % duty cycle for operator b(MCPWMXB), i.e for 48% duty cycle, duty_b = 48.0  
mcpwm_duty_type_t duty_mode  
    Set type of duty cycle  
mcpwm_counter_type_t counter_mode  
    Set type of MCPWM counter  
struct mcpwm_carrier_config_t  
    MCPWM config carrier structure.
```

Public Members

```
uint8_t carrier_period  
    Set carrier period = (carrier_period + 1)*800ns, carrier_period should be < 16  
uint8_t carrier_duty  
    Set carrier duty cycle, carrier_duty should be less than 8 (increment every 12.5%)  
uint8_t pulse_width_in_os  
    Set pulse width of first pulse in one shot mode = (carrier period)*(pulse_width_in_os + 1), should be less  
    then 16  
mcpwm_carrier_os_t carrier_os_mode  
    Enable or disable carrier oneshot mode  
mcpwm_carrier_out_ivt_t carrier_ivt_mode  
    Invert output of carrier
```

Macros

MCPWM_OPR_A

MCPWM_OPR_B

MCPWM_OPR_MAX

MCPWM_NO_CHANGE_IN_MCPWMXA

MCPWM_FORCE_MCPWMXA_LOW

MCPWM_FORCE_MCPWMXA_HIGH

MCPWM_TOG_MCPWMXA

MCPWM_NO_CHANGE_IN_MCPWMXB

MCPWM_FORCE_MCPWMXB_LOW

MCPWM_FORCE_MCPWMXB_HIGH

MCPWM_TOG_MCPWMXB

Type Definitions

```
typedef mcpwm_generator_t mcpwm_operator_t
```

```
typedef mcpwm_output_action_t mcpwm_action_on_pwmxa_t
```

```
typedef mcpwm_output_action_t mcpwm_action_on_pwmxb_t
```

Enumerations

```
enum mcpwm_io_signals_t
```

IO signals for the MCPWM.

- 6 MCPWM output pins that generate PWM signals
- 3 MCPWM fault input pins to detect faults like overcurrent, overvoltage, etc.
- 3 MCPWM sync input pins to synchronize MCPWM outputs signals
- 3 MCPWM capture input pins to gather feedback from controlled motors, using e.g. hall sensors

Values:

MCPWM0A = 0

PWM0A output pin

MCPWM0B

PWM0B output pin

MCPWM1A

PWM1A output pin

MCPWM1B

PWM1B output pin

MCPWM2A

PWM2A output pin

MCPWM2B

PWM2B output pin

MCPWM_SYNC_0

SYNC0 input pin

MCPWM_SYNC_1

SYNC1 input pin

MCPWM_SYNC_2

SYNC2 input pin

MCPWM_FAULT_0

FAULT0 input pin

MCPWM_FAULT_1
FAULT1 input pin

MCPWM_FAULT_2
FAULT2 input pin

MCPWM_CAP_0 = 84
CAP0 input pin

MCPWM_CAP_1
CAP1 input pin

MCPWM_CAP_2
CAP2 input pin

enum mcpwm_unit_t
Select MCPWM unit.

Values:

MCPWM_UNIT_0 = 0
MCPWM unit0 selected

MCPWM_UNIT_1
MCPWM unit1 selected

MCPWM_UNIT_MAX
Num of MCPWM units on ESP32

enum mcpwm_timer_t
Select MCPWM timer.

Values:

MCPWM_TIMER_0 = 0
Select MCPWM timer0

MCPWM_TIMER_1
Select MCPWM timer1

MCPWM_TIMER_2
Select MCPWM timer2

MCPWM_TIMER_MAX
Num of MCPWM timers on ESP32

enum mcpwm_generator_t
Select MCPWM operator.

Values:

MCPWM_GEN_A = 0
Select MCPWMXA, where 'X' is operator number

MCPWM_GEN_B
Select MCPWMXB, where 'X' is operator number

MCPWM_GEN_MAX
Num of generators to each operator of MCPWM

enum mcpwm_carrier_os_t
MCPWM carrier oneshot mode, in this mode the width of the first pulse of carrier can be programmed.

Values:

MCPWM_ONESHOT_MODE_DIS = 0
Enable oneshot mode

MCPWM_ONESHOT_MODE_EN
Disable oneshot mode

enum mcpwm_carrier_out_ivt_t

MCPWM carrier output inversion, high frequency carrier signal active with MCPWM signal is high.

Values:

MCPWM_CARRIER_OUT_IVT_DIS = 0

Enable carrier output inversion

MCPWM_CARRIER_OUT_IVT_EN

Disable carrier output inversion

enum mcpwm_fault_signal_t

MCPWM select fault signal input.

Values:

MCPWM_SELECT_F0 = 0

Select F0 as input

MCPWM_SELECT_F1

Select F1 as input

MCPWM_SELECT_F2

Select F2 as input

enum mcpwm_fault_input_level_t

MCPWM select triggering level of fault signal.

Values:

MCPWM_LOW_LEVEL_TGR = 0

Fault condition occurs when fault input signal goes from high to low, currently not supported

MCPWM_HIGH_LEVEL_TGR

Fault condition occurs when fault input signal goes low to high

enum mcpwm_capture_signal_t

MCPWM select capture signal input.

Values:

MCPWM_SELECT_CAP0 = 0

Select CAP0 as input

MCPWM_SELECT_CAP1

Select CAP1 as input

MCPWM_SELECT_CAP2

Select CAP2 as input

2.3.9 Pulse Counter

Introduction

The PCNT (Pulse Counter) module is designed to count the number of rising and/or falling edges of an input signal. Each pulse counter unit has a 16-bit signed counter register and two channels that can be configured to either increment or decrement the counter. Each channel has a signal input that accepts signal edges to be detected, as well as a control input that can be used to enable or disable the signal input. The inputs have optional filters that can be used to discard unwanted glitches in the signal.

Functionality Overview

Description of functionality of this API has been broken down into four sections:

- [Configuration](#) - describes counter's configuration parameters and how to setup the counter.

- *Operating the Counter* - provides information on control functions to pause, measure and clear the counter.
- *Filtering Pulses* - describes options to filtering pulses and the counter control signals.
- *Using Interrupts* - presents how to trigger interrupts on specific states of the counter.

Configuration

The PCNT module has eight independent counting “units” numbered from 0 to 7. In the API they are referred to using `pcnt_unit_t`. Each unit has two independent channels numbered as 0 and 1 and specified with `pcnt_channel_t`.

The configuration is provided separately per unit’s channel using `pcnt_config_t` and covers:

- The unit and the channel number this configuration refers to.
- GPIO numbers of the pulse input and the pulse gate input.
- Two pairs of parameters: `pcnt_ctrl_mode_t` and `pcnt_count_mode_t` to define how the counter reacts depending on the the status of control signal and how counting is done positive / negative edge of the pulses.
- Two limit values (minimum / maximum) that are used to establish watchpoints and trigger interrupts when the pulse count is meeting particular limit.

Setting up of particular channel is then done by calling a function `pcnt_unit_config()` with above `pcnt_config_t` as the input parameter.

To disable the pulse or the control input pin in configuration, provide `PCNT_PIN_NOT_USED` instead of the GPIO number.

Operating the Counter

After doing setup with `pcnt_unit_config()`, the counter immediately starts to operate. The accumulated pulse count can be checked by calling `pcnt_get_counter_value()`.

There are couple of functions that allow to control the counter’s operation: `pcnt_counter_pause()`, `pcnt_counter_resume()` and `pcnt_counter_clear()`

It is also possible to dynamically change the previously set up counter modes with `pcnt_unit_config()` by calling `pcnt_set_mode()`.

If desired, the pulse input pin and the control input pin may be changed “on the fly” using `pcnt_set_pin()`. To disable particular input provide as a function parameter `PCNT_PIN_NOT_USED` instead of the GPIO number.

Note: For the counter not to miss any pulses, the pulse duration should be longer than one APB_CLK cycle (12.5 ns). The pulses are sampled on the edges of the APB_CLK clock and may be missed, if fall between the edges. This applies to counter operation with or without a *filer*.

Filtering Pulses

The PCNT unit features filters on each of the pulse and control inputs, adding the option to ignore short glitches in the signals.

The length of ignored pulses is provided in APB_CLK clock cycles by calling `pcnt_set_filter_value()`. The current filter setting may be checked with `pcnt_get_filter_value()`. The APB_CLK clock is running at 80 MHz.

The filter is put into operation / suspended by calling `pcnt_filter_enable()` / `pcnt_filter_disable()`.

Using Interrupts

There are five counter state watch events, defined in `pcnt_evt_type_t`, that are able to trigger an interrupt. The event happens on the pulse counter reaching specific values:

- Minimum or maximum count values: `counter_l_lim` or `counter_h_lim` provided in `pcnt_config_t` as discussed in *Configuration*
- Threshold 0 or Threshold 1 values set using function `pcnt_set_event_value()`.
- Pulse count = 0

To register, enable or disable an interrupt to service the above events, call `pcnt_isr_register()`, `pcnt_intr_enable()`, and `pcnt_intr_disable()`. To enable or disable events on reaching threshold values, you will also need to call functions `pcnt_event_enable()` and `pcnt_event_disable()`.

In order to check what are the threshold values currently set, use function `pcnt_get_event_value()`.

Application Example

Pulse counter with control signal and event interrupt example: [peripherals/pcnt](#).

API Reference

Header File

- [driver/include/driver/pcnt.h](#)

Functions

`esp_err_t pcnt_unit_config(const pcnt_config_t *pcnt_config)`

Configure Pulse Counter unit.

Note This function will disable three events: `PCNT_EVT_L_LIM`, `PCNT_EVT_H_LIM`, `PCNT_EVT_ZERO`.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_STATE` pcnt driver already initialized
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `pcnt_config`: Pointer of Pulse Counter unit configure parameter

`esp_err_t pcnt_get_counter_value(pcnt_unit_t pcnt_unit, int16_t *count)`

Get pulse counter value.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_STATE` pcnt driver has not been initialized
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `pcnt_unit`: Pulse Counter unit number
- `count`: Pointer to accept counter value

`esp_err_t pcnt_counter_pause(pcnt_unit_t pcnt_unit)`

Pause PCNT counter of PCNT unit.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_STATE` pcnt driver has not been initialized
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `pcnt_unit`: PCNT unit number

esp_err_t **pcnt_counter_resume** (*pcnt_unit_t* *pcnt_unit*)

Resume counting for PCNT counter.

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE pcnt driver has not been initialized
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- *pcnt_unit*: PCNT unit number, select from *pcnt_unit_t*

esp_err_t **pcnt_counter_clear** (*pcnt_unit_t* *pcnt_unit*)

Clear and reset PCNT counter value to zero.

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE pcnt driver has not been initialized
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- *pcnt_unit*: PCNT unit number, select from *pcnt_unit_t*

esp_err_t **pcnt_intr_enable** (*pcnt_unit_t* *pcnt_unit*)

Enable PCNT interrupt for PCNT unit.

Note Each Pulse counter unit has five watch point events that share the same interrupt. Configure events with *pcnt_event_enable()* and *pcnt_event_disable()*

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE pcnt driver has not been initialized
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- *pcnt_unit*: PCNT unit number

esp_err_t **pcnt_intr_disable** (*pcnt_unit_t* *pcnt_unit*)

Disable PCNT interrupt for PCNT unit.

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE pcnt driver has not been initialized
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- *pcnt_unit*: PCNT unit number

esp_err_t **pcnt_event_enable** (*pcnt_unit_t* *unit*, *pcnt_evt_type_t* *evt_type*)

Enable PCNT event of PCNT unit.

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE pcnt driver has not been initialized
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- *unit*: PCNT unit number
- *evt_type*: Watch point event type. All enabled events share the same interrupt (one interrupt per pulse counter unit).

esp_err_t **pcnt_event_disable** (*pcnt_unit_t* *unit*, *pcnt_evt_type_t* *evt_type*)

Disable PCNT event of PCNT unit.

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE pcnt driver has not been initialized
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- *unit*: PCNT unit number

- `evt_type`: Watch point event type. All enabled events share the same interrupt (one interrupt per pulse counter unit).

`esp_err_t pcnt_set_event_value` (`pcnt_unit_t unit`, `pcnt_evt_type_t evt_type`, `int16_t value`)

Set PCNT event value of PCNT unit.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_STATE` pcnt driver has not been initialized
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `unit`: PCNT unit number
- `evt_type`: Watch point event type. All enabled events share the same interrupt (one interrupt per pulse counter unit).
- `value`: Counter value for PCNT event

`esp_err_t pcnt_get_event_value` (`pcnt_unit_t unit`, `pcnt_evt_type_t evt_type`, `int16_t *value`)

Get PCNT event value of PCNT unit.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_STATE` pcnt driver has not been initialized
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `unit`: PCNT unit number
- `evt_type`: Watch point event type. All enabled events share the same interrupt (one interrupt per pulse counter unit).
- `value`: Pointer to accept counter value for PCNT event

`esp_err_t pcnt_isr_register` (`void (*fn)`) `void *`

, `void *arg`, `int intr_alloc_flags`, `pcnt_isr_handle_t *handle` Register PCNT interrupt handler, the handler is an ISR. The handler will be attached to the same CPU core that this function is running on. Please do not use `pcnt_isr_service_install` if this function was called.

Return

- `ESP_OK` Success
- `ESP_ERR_NOT_FOUND` Can not find the interrupt that matches the flags.
- `ESP_ERR_INVALID_ARG` Function pointer error.

Parameters

- `fn`: Interrupt handler function.
- `arg`: Parameter for handler function
- `intr_alloc_flags`: Flags used to allocate the interrupt. One or multiple (ORred) `ESP_INTR_FLAG_*` values. See `esp_intr_alloc.h` for more info.
- `handle`: Pointer to return handle. If non-NULL, a handle for the interrupt will be returned here. Calling `esp_intr_free` to unregister this ISR service if needed, but only if the handle is not NULL.

`esp_err_t pcnt_set_pin` (`pcnt_unit_t unit`, `pcnt_channel_t channel`, `int pulse_io`, `int ctrl_io`)

Configure PCNT pulse signal input pin and control input pin.

Note Set the signal input to `PCNT_PIN_NOT_USED` if unused.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_STATE` pcnt driver has not been initialized
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `unit`: PCNT unit number
- `channel`: PCNT channel number
- `pulse_io`: Pulse signal input GPIO
- `ctrl_io`: Control signal input GPIO

`esp_err_t pcnt_filter_enable` (`pcnt_unit_t unit`)

Enable PCNT input filter.

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE pcnt driver has not been initialized
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `unit`: PCNT unit number

esp_err_t **pcnt_filter_disable** (*pcnt_unit_t* `unit`)

Disable PCNT input filter.

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE pcnt driver has not been initialized
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `unit`: PCNT unit number

esp_err_t **pcnt_set_filter_value** (*pcnt_unit_t* `unit`, *uint16_t* `filter_val`)

Set PCNT filter value.

Note `filter_val` is a 10-bit value, so the maximum `filter_val` should be limited to 1023.

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE pcnt driver has not been initialized
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `unit`: PCNT unit number
- `filter_val`: PCNT signal filter value, counter in APB_CLK cycles. Any pulses lasting shorter than this will be ignored when the filter is enabled.

esp_err_t **pcnt_get_filter_value** (*pcnt_unit_t* `unit`, *uint16_t* *`filter_val`)

Get PCNT filter value.

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE pcnt driver has not been initialized
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `unit`: PCNT unit number
- `filter_val`: Pointer to accept PCNT filter value.

esp_err_t **pcnt_set_mode** (*pcnt_unit_t* `unit`, *pcnt_channel_t* `channel`, *pcnt_count_mode_t* `pos_mode`, *pcnt_count_mode_t* `neg_mode`, *pcnt_ctrl_mode_t* `hctrl_mode`, *pcnt_ctrl_mode_t* `lctrl_mode`)

Set PCNT counter mode.

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE pcnt driver has not been initialized
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `unit`: PCNT unit number
- `channel`: PCNT channel number
- `pos_mode`: Counter mode when detecting positive edge
- `neg_mode`: Counter mode when detecting negative edge
- `hctrl_mode`: Counter mode when control signal is high level
- `lctrl_mode`: Counter mode when control signal is low level

esp_err_t **pcnt_isr_handler_add** (*pcnt_unit_t* `unit`, *void* (*`isr_handler`)) *void* *`args`

Add ISR handler for specified unit.

Call this function after using `pcnt_isr_service_install()` to install the PCNT driver's ISR handler service.

The ISR handlers do not need to be declared with `IRAM_ATTR`, unless you pass the `ESP_INTR_FLAG_IRAM` flag when allocating the ISR in `pcnt_isr_service_install()`.

This ISR handler will be called from an ISR. So there is a stack size limit (configurable as “ISR stack size” in `menuconfig`). This limit is smaller compared to a global PCNT interrupt handler due to the additional level of indirection.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_STATE` pcnt driver has not been initialized
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `unit`: PCNT unit number
- `isr_handler`: Interrupt handler function.
- `args`: Parameter for handler function

esp_err_t `pcnt_isr_service_install` (int *intr_alloc_flags*)

Install PCNT ISR service.

Note We can manage different interrupt service for each unit. This function will use the default ISR handle service, Calling `pcnt_isr_service_uninstall` to uninstall the default service if needed. Please do not use `pcnt_isr_register` if this function was called.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_STATE` pcnt driver has not been initialized
- `ESP_ERR_NO_MEM` No memory to install this service
- `ESP_ERR_INVALID_STATE` ISR service already installed

Parameters

- `intr_alloc_flags`: Flags used to allocate the interrupt. One or multiple (ORred) `ESP_INTR_FLAG_*` values. See `esp_intr_alloc.h` for more info.

void `pcnt_isr_service_uninstall` (void)

Uninstall PCNT ISR service, freeing related resources.

esp_err_t `pcnt_isr_handler_remove` (*pcnt_unit_t* *unit*)

Delete ISR handler for specified unit.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_STATE` pcnt driver has not been initialized
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `unit`: PCNT unit number

Type Definitions

```
typedef intr_handle_t pcnt_isr_handle_t
```

Header File

- `soc/include/hal/pcnt_types.h`

Structures

```
struct pcnt_config_t
```

Pulse Counter configuration for a single channel.

Public Members

```
int pulse_gpio_num
```

Pulse input GPIO number, if you want to use GPIO16, enter `pulse_gpio_num = 16`, a negative value will

be ignored

int **ctrl_gpio_num**

Control signal input GPIO number, a negative value will be ignored

pcnt_ctrl_mode_t **lctrl_mode**

PCNT low control mode

pcnt_ctrl_mode_t **hctrl_mode**

PCNT high control mode

pcnt_count_mode_t **pos_mode**

PCNT positive edge count mode

pcnt_count_mode_t **neg_mode**

PCNT negative edge count mode

int16_t **counter_h_lim**

Maximum counter value

int16_t **counter_l_lim**

Minimum counter value

pcnt_unit_t **unit**

PCNT unit number

pcnt_channel_t **channel**

the PCNT channel

Type Definitions

typedef int **pcnt_port_t**

PCNT port number, the max port number is (PCNT_PORT_MAX - 1).

typedef int **pcnt_unit_t**

Selection of all available PCNT units.

Enumerations

enum **pcnt_ctrl_mode_t**

Selection of available modes that determine the counter's action depending on the state of the control signal's input GPIO.

Note Configuration covers two actions, one for high, and one for low level on the control input

Values:

PCNT_MODE_KEEP = 0

Control mode: won't change counter mode

PCNT_MODE_REVERSE = 1

Control mode: invert counter mode(increase -> decrease, decrease -> increase)

PCNT_MODE_DISABLE = 2

Control mode: Inhibit counter(counter value will not change in this condition)

PCNT_MODE_MAX

enum **pcnt_count_mode_t**

Selection of available modes that determine the counter's action on the edge of the pulse signal's input GPIO.

Note Configuration covers two actions, one for positive, and one for negative edge on the pulse input

Values:

PCNT_COUNT_DIS = 0

Counter mode: Inhibit counter(counter value will not change in this condition)

PCNT_COUNT_INC = 1

Counter mode: Increase counter value

PCNT_COUNT_DEC = 2
Counter mode: Decrease counter value

PCNT_COUNT_MAX

enum pcnt_channel_t
Selection of channels available for a single PCNT unit.

Values:

PCNT_CHANNEL_0 = 0x00
PCNT channel 0

PCNT_CHANNEL_1 = 0x01
PCNT channel 1

PCNT_CHANNEL_MAX

enum pcnt_evt_type_t
Selection of counter's events that may trigger an interrupt.

Values:

PCNT_EVT_THRES_1 = BIT(2)
PCNT watch point event: threshold1 value event

PCNT_EVT_THRES_0 = BIT(3)
PCNT watch point event: threshold0 value event

PCNT_EVT_L_LIM = BIT(4)
PCNT watch point event: Minimum counter value

PCNT_EVT_H_LIM = BIT(5)
PCNT watch point event: Maximum counter value

PCNT_EVT_ZERO = BIT(6)
PCNT watch point event: counter value zero event

PCNT_EVT_MAX

2.3.10 RMT

The RMT (Remote Control) module driver can be used to send and receive infrared remote control signals. Due to the flexibility of the RMT module, the driver can also be used to generate or receive many other types of signals.

The signal, which consists of a series of pulses, is generated by the RMT's transmitter based on a list of values. The values define the pulse duration and a binary level, see below. The transmitter can also provide a carrier and modulate it with provided pulses.

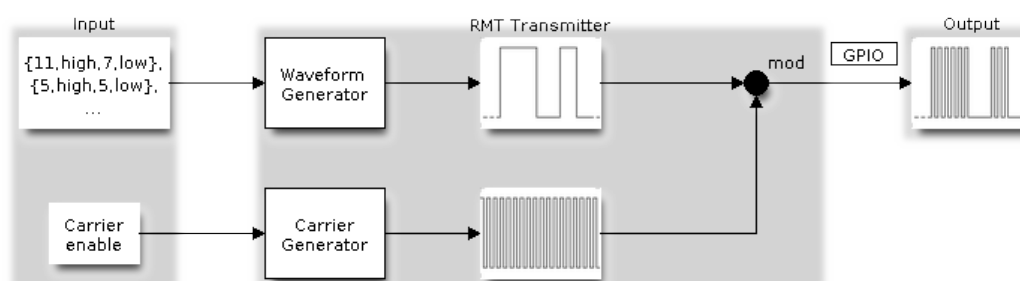


Fig. 19: RMT Transmitter Overview

The reverse operation is performed by the receiver, where a series of pulses is decoded into a list of values containing the pulse duration and binary level. A filter may be applied to remove high frequency noise from the input signal.

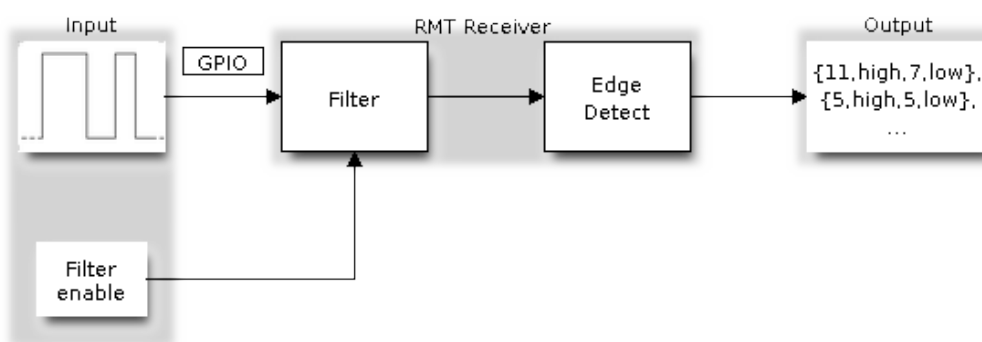


Fig. 20: RMT Receiver Overview

There couple of typical steps to setup and operate the RMT and they are discussed in the following sections:

1. *Configure Driver*
2. *Transmit Data or Receive Data*
3. *Change Operation Parameters*
4. *Use Interrupts*

The RMT has eight channels numbered from zero to seven. Each channel is able to independently transmit or receive data. They are referred to using indexes defined in structure `rmt_channel_t`.

Configure Driver

There are several parameters that define how particular channel operates. Most of these parameters are configured by setting specific members of `rmt_config_t` structure. Some of the parameters are common to both transmit or receive mode, and some are mode specific. They are all discussed below.

Common Parameters

- The **channel** to be configured, select one from the `rmt_channel_t` enumerator.
- The RMT **operation mode** - whether this channel is used to transmit or receive data, selected by setting a **rmt_mode** members to one of the values from `rmt_mode_t`.
- What is the **pin number** to transmit or receive RMT signals, selected by setting **gpio_num**.
- How many **memory blocks** will be used by the channel, set with **mem_block_num**.
- A **clock divider**, that will determine the range of pulse length generated by the RMT transmitter or discriminated by the receiver. Selected by setting **clk_div** to a value within [1 .. 255] range. The RMT source clock is typically APB CLK, 80Mhz by default.

Note: The period of a square wave after the clock divider is called a ‘tick’ . The length of the pulses generated by the RMT transmitter or discriminated by the receiver is configured in number of ‘ticks’ .

There are also couple of specific parameters that should be set up depending if selected channel is configured in *Transmit Mode* or *Receive Mode*:

Transmit Mode When configuring channel in transmit mode, set **tx_config** and the following members of `rmt_tx_config_t`:

- Transmit the currently configured data items in a loop - **loop_en**
- Enable the RMT carrier signal - **carrier_en**
- Frequency of the carrier in Hz - **carrier_freq_hz**

- Duty cycle of the carrier signal in percent (%) - **carrier_duty_percent**
- Level of the RMT output, when the carrier is applied - **carrier_level**
- Enable the RMT output if idle - **idle_output_en**
- Set the signal level on the RMT output if idle - **idle_level**

Receive Mode In receive mode, set **rx_config** and the following members of `rmt_rx_config_t`:

- Enable a filter on the input of the RMT receiver - **filter_en**
- A threshold of the filter, set in the number of ticks - **filter_ticks_thresh**. Pulses shorter than this setting will be filtered out. Note, that the range of entered tick values is [0..255].
- A pulse length threshold that will turn the RMT receiver idle, set in number of ticks - **idle_threshold**. The receiver will ignore pulses longer than this setting.

Finalize Configuration Once the `rmt_config_t` structure is populated with parameters, it should be then invoked with `rmt_config()` to make the configuration effective.

The last configuration step is installation of the driver in memory by calling `rmt_driver_install()`. If `rx_buf_size` parameter of this function is > 0 , then a ring buffer for incoming data will be allocated. A default ISR handler will be installed, see a note in [Use Interrupts](#).

Now, depending on how the channel is configured, we are ready to either *Transmit Data* or *Receive Data*. This is described in next two sections.

Transmit Data

Before being able to transmit some RMT pulses, we need to define the pulse pattern. The minimum pattern recognized by the RMT controller, later called an ‘item’, is provided in a structure `rmt_item32_t`, see [soc/esp32/include/soc/rmt_caps.h](#). Each item consists of two pairs of two values. The first value in a pair describes the signal duration in ticks and is 15 bits long, the second provides the signal level (high or low) and is contained in a single bit. A block of couple of items and the structure of an item is presented below.

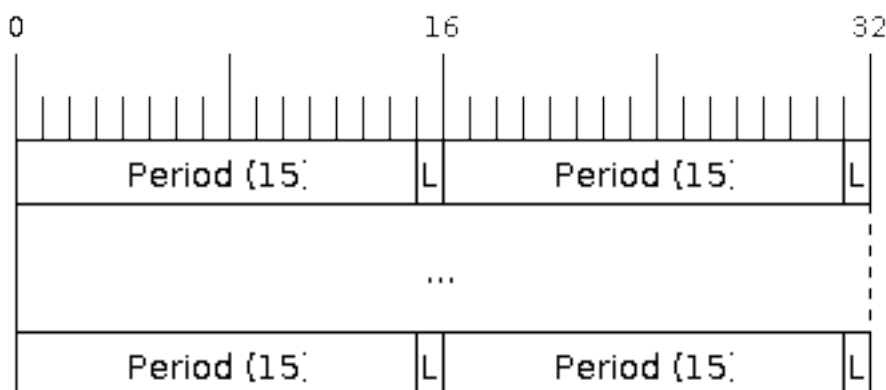


Fig. 21: Structure of RMT items (L - signal level)

For a simple example how to define a block of items see [peripherals/rmt/morse_code](#).

The items are provided to the RMT controller by calling function `rmt_write_items()`. This function also automatically triggers start of transmission. It may be called to wait for transmission completion or exit just after transmission start. In such case you can wait for the transmission end by calling `rmt_wait_tx_done()`. This

function does not limit the number of data items to transmit. It is using an interrupt to successively copy the new data chunks to RMT's internal memory as previously provided data are sent out.

Another way to provide data for transmission is by calling `rmt_fill_tx_items()`. In this case transmission is not started automatically. To control the transmission process use `rmt_tx_start()` and `rmt_tx_stop()`. The number of items to sent is restricted by the size of memory blocks allocated in the RMT controller's internal memory, see `rmt_set_mem_block_num()`.

Receive Data

Before starting the receiver we need some storage for incoming items. The RMT controller has 512 x 32-bits of internal RAM shared between all eight channels. In typical scenarios it is not enough as an ultimate storage for all incoming (and outgoing) items. Therefore this API supports retrieval of incoming items on the fly to save them in a ring buffer of a size defined by the user. The size is provided when calling `rmt_driver_install()` discussed above. To get a handle to this buffer call `rmt_get_ringbuf_handle()`.

With the above steps complete we can start the receiver by calling `rmt_rx_start()` and then move to checking what's inside the buffer. To do so, you can use common FreeRTOS functions that interact with the ring buffer. Please see an example how to do it in [peripherals/rmt/ir_protocols](#).

To stop the receiver, call `rmt_rx_stop()`.

Change Operation Parameters

Previously described function `rmt_config()` provides a convenient way to set several configuration parameters in one shot. This is usually done on application start. Then, when the application is running, the API provides an alternate way to update individual parameters by calling dedicated functions. Each function refers to the specific RMT channel provided as the first input parameter. Most of the functions have `_get_` counterpart to read back the currently configured value.

Parameters Common to Transmit and Receive Mode

- Selection of a GPIO pin number on the input or output of the RMT - `rmt_set_pin()`
- Number of memory blocks allocated for the incoming or outgoing data - `rmt_set_mem_pd()`
- Setting of the clock divider - `rmt_set_clk_div()`
- Selection of the clock source, note that currently one clock source is supported, the APB clock which is 80Mhz - `rmt_set_source_clk()`

Transmit Mode Parameters

- Enable or disable the loop back mode for the transmitter - `rmt_set_tx_loop_mode()`
- Binary level on the output to apply the carrier - `rmt_set_tx_carrier()`, selected from `rmt_carrier_level_t`
- Determines the binary level on the output when transmitter is idle - `rmt_set_idle_level()`, selected from `rmt_idle_level_t`

Receive Mode Parameters

- The filter setting - `rmt_set_rx_filter()`
- The receiver threshold setting - `rmt_set_rx_idle_thresh()`
- Whether the transmitter or receiver is entitled to access RMT's memory - `rmt_set_memory_owner()`, selection is from `rmt_mem_owner_t`.

Use Interrupts

Registering of an interrupt handler for the RMT controller is done by calling `rmt_isr_register()`.

Note: When calling `rmt_driver_install()` to use the system RMT driver, a default ISR is being installed. In such a case you cannot register a generic ISR handler with `rmt_isr_register()`.

The RMT controller triggers interrupts on four specific events described below. To enable interrupts on these events, the following functions are provided:

- The RMT receiver has finished receiving a signal - `rmt_set_rx_intr_en()`
- The RMT transmitter has finished transmitting the signal - `rmt_set_tx_intr_en()`
- The number of events the transmitter has sent matches a threshold value `rmt_set_tx_thr_intr_en()`
- Ownership to the RMT memory block has been violated - `rmt_set_err_intr_en()`

Setting or clearing an interrupt enable mask for specific channels and events may be also done by calling `rmt_set_intr_enable_mask()` or `rmt_clr_intr_enable_mask()`.

When servicing an interrupt within an ISR, the interrupt need to explicitly cleared. To do so, set specific bits described as `RMT.int_clr.val.chN_event_name` and defined as a volatile struct in `soc/esp32/include/soc/rmt_struct.h`, where N is the RMT channel number [0, 7] and the `event_name` is one of four events described above.

If you do not need an ISR anymore, you can deregister it by calling a function `rmt_isr_deregister()`.

Uninstall Driver

If the RMT driver has been installed with `rmt_driver_install()` for some specific period of time and then not required, the driver may be removed to free allocated resources by calling `rmt_driver_uninstall()`.

Application Examples

- A simple RMT TX example: [peripherals/rmt/morse_code](#).
- Another RMT TX example, specific to drive a common RGB LED strip: [peripherals/rmt/led_strip](#).
- NEC remote control TX and RX example: [peripherals/rmt/ir_protocols](#).

API Reference

Header File

- `driver/include/driver/rmt.h`

Functions

`esp_err_t rmt_set_clk_div(rmt_channel_t channel, uint8_t div_cnt)`

Set RMT clock divider, channel clock is divided from source clock.

Return

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

Parameters

- `channel`: RMT channel
- `div_cnt`: RMT counter clock divider

`esp_err_t rmt_get_clk_div(rmt_channel_t channel, uint8_t *div_cnt)`

Get RMT clock divider, channel clock is divided from source clock.

Return

- `ESP_ERR_INVALID_ARG` Parameter error

- ESP_OK Success

Parameters

- `channel`: RMT channel
- `div_cnt`: pointer to accept RMT counter divider

esp_err_t **rmt_set_rx_idle_thresh** (*rmt_channel_t* channel, uint16_t thresh)

Set RMT RX idle threshold value.

In receive mode, when no edge is detected on the input signal for longer than `idle_thres` channel clock cycles, the receive process is finished.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- `channel`: RMT channel
- `thresh`: RMT RX idle threshold

esp_err_t **rmt_get_rx_idle_thresh** (*rmt_channel_t* channel, uint16_t *thresh)

Get RMT idle threshold value.

In receive mode, when no edge is detected on the input signal for longer than `idle_thres` channel clock cycles, the receive process is finished.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- `channel`: RMT channel
- `thresh`: pointer to accept RMT RX idle threshold value

esp_err_t **rmt_set_mem_block_num** (*rmt_channel_t* channel, uint8_t rmt_mem_num)

Set RMT memory block number for RMT channel.

This function is used to configure the amount of memory blocks allocated to channel `n`. The 8 channels share a 512x32-bit RAM block which can be read and written by the processor cores over the APB bus, as well as read by the transmitters and written by the receivers.

The RAM address range for channel `n` is `start_addr_CHn` to `end_addr_CHn`, which are defined by: Memory block start address is `RMT_CHANNEL_MEM(n)` (in `soc/rmt_reg.h`), that is, `start_addr_chn = RMT base address + 0x800 + 64 * 4 * n`, and `end_addr_chn = RMT base address + 0x800 + 64 * 4 * n + 64 * 4 * RMT_MEM_SIZE_CHn mod 512 * 4`

Note If memory block number of one channel is set to a value greater than 1, this channel will occupy the memory block of the next channel. Channel 0 can use at most 8 blocks of memory, accordingly channel 7 can only use one memory block.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- `channel`: RMT channel
- `rmt_mem_num`: RMT RX memory block number, one block has $64 * 32$ bits.

esp_err_t **rmt_get_mem_block_num** (*rmt_channel_t* channel, uint8_t *rmt_mem_num)

Get RMT memory block number.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- `channel`: RMT channel
- `rmt_mem_num`: Pointer to accept RMT RX memory block number

esp_err_t **rmt_set_tx_carrier** (*rmt_channel_t* channel, bool carrier_en, uint16_t high_level, uint16_t low_level, *rmt_carrier_level_t* carrier_level)

Configure RMT carrier for TX signal.

Set different values for carrier_high and carrier_low to set different frequency of carrier. The unit of carrier_high/low is the source clock tick, not the divided channel counter clock.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- channel: RMT channel
- carrier_en: Whether to enable output carrier.
- high_level: High level duration of carrier
- low_level: Low level duration of carrier.
- carrier_level: Configure the way carrier wave is modulated for channel.
 - 1' b1:transmit on low output level
 - 1' b0:transmit on high output level

esp_err_t **rmt_set_mem_pd** (*rmt_channel_t* channel, bool pd_en)

Set RMT memory in low power mode.

Reduce power consumed by memory. 1:memory is in low power state.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- channel: RMT channel
- pd_en: RMT memory low power enable.

esp_err_t **rmt_get_mem_pd** (*rmt_channel_t* channel, bool *pd_en)

Get RMT memory low power mode.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- channel: RMT channel
- pd_en: Pointer to accept RMT memory low power mode.

esp_err_t **rmt_tx_start** (*rmt_channel_t* channel, bool tx_idx_rst)

Set RMT start sending data from memory.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- channel: RMT channel
- tx_idx_rst: Set true to reset memory index for TX. Otherwise, transmitter will continue sending from the last index in memory.

esp_err_t **rmt_tx_stop** (*rmt_channel_t* channel)

Set RMT stop sending.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- channel: RMT channel

esp_err_t **rmt_rx_start** (*rmt_channel_t* channel, bool rx_idx_rst)

Set RMT start receiving data.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- `channel`: RMT channel
- `rx_idx_rst`: Set true to reset memory index for receiver. Otherwise, receiver will continue receiving data to the last index in memory.

esp_err_t **rmt_rx_stop** (*rmt_channel_t* channel)

Set RMT stop receiving data.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- `channel`: RMT channel

esp_err_t **rmt_memory_rw_rst** (*rmt_channel_t* channel)

Reset RMT TX/RX memory index.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- `channel`: RMT channel

esp_err_t **rmt_set_memory_owner** (*rmt_channel_t* channel, *rmt_mem_owner_t* owner)

Set RMT memory owner.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- `channel`: RMT channel
- `owner`: To set when the transmitter or receiver can process the memory of channel.

esp_err_t **rmt_get_memory_owner** (*rmt_channel_t* channel, *rmt_mem_owner_t* *owner)

Get RMT memory owner.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- `channel`: RMT channel
- `owner`: Pointer to get memory owner.

esp_err_t **rmt_set_tx_loop_mode** (*rmt_channel_t* channel, bool loop_en)

Set RMT tx loop mode.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- `channel`: RMT channel
- `loop_en`: Enable RMT transmitter loop sending mode. If set true, transmitter will continue sending from the first data to the last data in channel over and over again in a loop.

esp_err_t **rmt_get_tx_loop_mode** (*rmt_channel_t* channel, bool *loop_en)

Get RMT tx loop mode.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- `channel`: RMT channel

- `loop_en`: Pointer to accept RMT transmitter loop sending mode.

esp_err_t **rmt_set_rx_filter** (*rmt_channel_t* channel, bool *rx_filter_en*, uint8_t *thresh*)
Set RMT RX filter.

In receive mode, channel will ignore input pulse when the pulse width is smaller than threshold. Counted in source clock, not divided counter clock.

Return

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

Parameters

- `channel`: RMT channel
- `rx_filter_en`: To enable RMT receiver filter.
- `thresh`: Threshold of pulse width for receiver.

esp_err_t **rmt_set_source_clk** (*rmt_channel_t* channel, *rmt_source_clk_t* base_clk)
Set RMT source clock.

RMT module has two clock sources:

1. APB clock which is 80Mhz
2. REF tick clock, which would be 1Mhz (not supported in this version).

Return

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

Parameters

- `channel`: RMT channel
- `base_clk`: To choose source clock for RMT module.

esp_err_t **rmt_get_source_clk** (*rmt_channel_t* channel, *rmt_source_clk_t* **src_clk*)
Get RMT source clock.

RMT module has two clock sources:

1. APB clock which is 80Mhz
2. REF tick clock, which would be 1Mhz (not supported in this version).

Return

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

Parameters

- `channel`: RMT channel
- `src_clk`: Pointer to accept source clock for RMT module.

esp_err_t **rmt_set_idle_level** (*rmt_channel_t* channel, bool *idle_out_en*, *rmt_idle_level_t* level)
Set RMT idle output level for transmitter.

Return

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

Parameters

- `channel`: RMT channel
- `idle_out_en`: To enable idle level output.
- `level`: To set the output signal's level for channel in idle state.

esp_err_t **rmt_get_idle_level** (*rmt_channel_t* channel, bool **idle_out_en*, *rmt_idle_level_t* **level*)
Get RMT idle output level for transmitter.

Return

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

Parameters

- `channel`: RMT channel
- `idle_out_en`: Pointer to accept value of enable idle.

- `level`: Pointer to accept value of output signal's level in idle state for specified channel.

esp_err_t **rmt_get_status** (*rmt_channel_t* channel, uint32_t *status)

Get RMT status.

Return

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

Parameters

- `channel`: RMT channel
- `status`: Pointer to accept channel status. Please refer to `RMT_CHnSTATUS_REG(n=0~7)` in `rmt_reg.h` for more details of each field.

void **rmt_set_intr_enable_mask** (uint32_t mask)

Set mask value to RMT interrupt enable register.

Parameters

- `mask`: Bit mask to set to the register

void **rmt_clr_intr_enable_mask** (uint32_t mask)

Clear mask value to RMT interrupt enable register.

Parameters

- `mask`: Bit mask to clear the register

esp_err_t **rmt_set_rx_intr_en** (*rmt_channel_t* channel, bool en)

Set RMT RX interrupt enable.

Return

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

Parameters

- `channel`: RMT channel
- `en`: enable or disable RX interrupt.

esp_err_t **rmt_set_err_intr_en** (*rmt_channel_t* channel, bool en)

Set RMT RX error interrupt enable.

Return

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

Parameters

- `channel`: RMT channel
- `en`: enable or disable RX err interrupt.

esp_err_t **rmt_set_tx_intr_en** (*rmt_channel_t* channel, bool en)

Set RMT TX interrupt enable.

Return

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

Parameters

- `channel`: RMT channel
- `en`: enable or disable TX interrupt.

esp_err_t **rmt_set_tx_thr_intr_en** (*rmt_channel_t* channel, bool en, uint16_t evt_thresh)

Set RMT TX threshold event interrupt enable.

An interrupt will be triggered when the number of transmitted items reaches the threshold value

Return

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

Parameters

- `channel`: RMT channel
- `en`: enable or disable TX event interrupt.

- `evt_thresh`: RMT event interrupt threshold value

`esp_err_t rmt_set_pin(rmt_channel_t channel, rmt_mode_t mode, gpio_num_t gpio_num)`

Set RMT pin.

Return

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

Parameters

- `channel`: RMT channel
- `mode`: TX or RX mode for RMT
- `gpio_num`: GPIO number to transmit or receive the signal.

`esp_err_t rmt_config(const rmt_config_t *rmt_param)`

Configure RMT parameters.

Return

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

Parameters

- `rmt_param`: RMT parameter struct

`esp_err_t rmt_isr_register(void (*fn)) void *`

`, void *arg, int intr_alloc_flags, rmt_isr_handle_t *handle` Register RMT interrupt handler, the handler is an ISR.

The handler will be attached to the same CPU core that this function is running on.

Note If you already called `rmt_driver_install` to use system RMT driver, please do not register ISR handler again.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Function pointer error.
- `ESP_FAIL` System driver installed, can not register ISR handler for RMT

Parameters

- `fn`: Interrupt handler function.
- `arg`: Parameter for the handler function
- `intr_alloc_flags`: Flags used to allocate the interrupt. One or multiple (ORred) `ESP_INTR_FLAG_*` values. See `esp_intr_alloc.h` for more info.
- `handle`: If non-zero, a handle to later clean up the ISR gets stored here.

`esp_err_t rmt_isr_deregister(rmt_isr_handle_t handle)`

Deregister previously registered RMT interrupt handler.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Handle invalid

Parameters

- `handle`: Handle obtained from `rmt_isr_register`

`esp_err_t rmt_fill_tx_items(rmt_channel_t channel, const rmt_item32_t *item, uint16_t item_num, uint16_t mem_offset)`

Fill memory data of channel with given RMT items.

Return

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

Parameters

- `channel`: RMT channel
- `item`: Pointer of items.
- `item_num`: RMT sending items number.
- `mem_offset`: Index offset of memory.

esp_err_t **rmt_driver_install** (*rmt_channel_t* channel, size_t rx_buf_size, int intr_alloc_flags)

Initialize RMT driver.

Return

- ESP_ERR_INVALID_STATE Driver is already installed, call rmt_driver_uninstall first.
- ESP_ERR_NO_MEM Memory allocation failure
- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- channel: RMT channel
- rx_buf_size: Size of RMT RX ringbuffer. Can be 0 if the RX ringbuffer is not used.
- intr_alloc_flags: Flags for the RMT driver interrupt handler. Pass 0 for default flags. See esp_intr_alloc.h for details. If ESP_INTR_FLAG_IRAM is used, please do not use the memory allocated from psram when calling rmt_write_items.

esp_err_t **rmt_driver_uninstall** (*rmt_channel_t* channel)

Uninstall RMT driver.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- channel: RMT channel

esp_err_t **rmt_get_channel_status** (*rmt_channel_status_result_t* *channel_status)

Get the current status of eight channels.

Note Do not call this function if it is possible that rmt_driver_uninstall will be called at the same time.

Return

- ESP_ERR_INVALID_ARG Parameter is NULL
- ESP_OK Success

Parameters

- [out] channel_status: store the current status of each channel

esp_err_t **rmt_get_counter_clock** (*rmt_channel_t* channel, uint32_t *clock_hz)

Get speed of channel's internal counter clock.

Return

- ESP_ERR_INVALID_ARG Parameter is NULL
- ESP_OK Success

Parameters

- channel: RMT channel
- [out] clock_hz: counter clock speed, in hz

esp_err_t **rmt_write_items** (*rmt_channel_t* channel, const rmt_item32_t *rmt_item, int item_num, bool wait_tx_done)

RMT send waveform from rmt_item array.

This API allows user to send waveform with any length.

Note This function will not copy data, instead, it will point to the original items, and send the waveform items. If wait_tx_done is set to true, this function will block and will not return until all items have been sent out. If wait_tx_done is set to false, this function will return immediately, and the driver interrupt will continue sending the items. We must make sure the item data will not be damaged when the driver is still sending items in driver interrupt.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- channel: RMT channel
- rmt_item: head point of RMT items array. If ESP_INTR_FLAG_IRAM is used, please do not use the memory allocated from psram when calling rmt_write_items.

- `item_num`: RMT data item number.
- `wait_tx_done`:
 - If set 1, it will block the task and wait for sending done.
 - If set 0, it will not wait and return immediately.

esp_err_t **rmt_wait_tx_done** (*rmt_channel_t* channel, TickType_t wait_time)
Wait RMT TX finished.

Return

- ESP_OK RMT Tx done successfully
- ESP_ERR_TIMEOUT Exceeded the 'wait_time' given
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Driver not installed

Parameters

- `channel`: RMT channel
- `wait_time`: Maximum time in ticks to wait for transmission to be complete. If set 0, return immediately with ESP_ERR_TIMEOUT if TX is busy (polling).

esp_err_t **rmt_get_ringbuf_handle** (*rmt_channel_t* channel, RingbufHandle_t *buf_handle)
Get ringbuffer from RMT.

Users can get the RMT RX ringbuffer handle, and process the RX data.

Return

- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

Parameters

- `channel`: RMT channel
- `buf_handle`: Pointer to buffer handle to accept RX ringbuffer handle.

esp_err_t **rmt_translator_init** (*rmt_channel_t* channel, *sample_to_rmt_t* fn)

Init rmt translator and register user callback. The callback will convert the raw data that needs to be sent to rmt format. If a channel is initialized more than once, the user callback will be replaced by the later.

Return

- ESP_FAIL Init fail.
- ESP_OK Init success.

Parameters

- `channel`: RMT channel .
- `fn`: Point to the data conversion function.

esp_err_t **rmt_write_sample** (*rmt_channel_t* channel, **const** uint8_t *src, size_t src_size, bool wait_tx_done)

Translate uint8_t type of data into rmt format and send it out. Requires `rmt_translator_init` to init the translator first.

Return

- ESP_FAIL Send fail
- ESP_OK Send success

Parameters

- `channel`: RMT channel .
- `src`: Pointer to the raw data.
- `src_size`: The size of the raw data.
- `wait_tx_done`: Set true to wait all data send done.

rmt_tx_end_callback_t **rmt_register_tx_end_callback** (*rmt_tx_end_fn_t* function, void *arg)

Registers a callback that will be called when transmission ends.

Called by `rmt_driver_isr_default` in interrupt context.

Note Requires `rmt_driver_install` to install the default ISR handler.

Return the previous callback settings (members will be set to NULL if there was none)

Parameters

- `function`: Function to be called from the default interrupt handler or NULL.

- `arg`: Argument which will be provided to the callback when it is called.

Structures

struct rmt_tx_config_t

Data struct of RMT TX configure parameters.

Public Members

`uint32_t carrier_freq_hz`

RMT carrier frequency

`rmt_carrier_level_t carrier_level`

Level of the RMT output, when the carrier is applied

`rmt_idle_level_t idle_level`

RMT idle level

`uint8_t carrier_duty_percent`

RMT carrier duty (%)

`bool carrier_en`

RMT carrier enable

`bool loop_en`

Enable sending RMT items in a loop

`bool idle_output_en`

RMT idle level output enable

struct rmt_rx_config_t

Data struct of RMT RX configure parameters.

Public Members

`uint16_t idle_threshold`

RMT RX idle threshold

`uint8_t filter_ticks_thresh`

RMT filter tick number

`bool filter_en`

RMT receiver filter enable

struct rmt_config_t

Data struct of RMT configure parameters.

Public Members

`rmt_mode_t rmt_mode`

RMT mode: transmitter or receiver

`rmt_channel_t channel`

RMT channel

`gpio_num_t gpio_num`

RMT GPIO number

`uint8_t clk_div`

RMT channel counter divider

`uint8_t mem_block_num`

RMT memory block number

rmt_tx_config_t **tx_config**

RMT TX parameter

rmt_rx_config_t **rx_config**

RMT RX parameter

struct rmt_tx_end_callback_t

Structure encapsulating a RMT TX end callback.

Public Members

rmt_tx_end_fn_t **function**

Function which is called on RMT TX end

void ***arg**

Optional argument passed to function

Macros

RMT_MEM_ITEM_NUM

Define memory space of each RMT channel (in words = 4 bytes)

RMT_DEFAULT_CONFIG_TX (gpio, channel_id)

Default configuration for Tx channel.

RMT_DEFAULT_CONFIG_RX (gpio, channel_id)

Default configuration for RX channel.

Type Definitions

typedef intr_handle_t rmt_isr_handle_t

RMT interrupt handle.

typedef void (*rmt_tx_end_fn_t) (rmt_channel_t channel, void *arg)

Type of RMT Tx End callback function.

typedef void (*sample_to_rmt_t) (const void *src, rmt_item32_t *dest, size_t src_size, size_t wanted_num, size_t *translated_size, size_t *item_num)

User callback function to convert uint8_t type data to rmt format(rmt_item32_t).

This function may be called from an ISR, so, the code should be short and efficient.

Note In fact, item_num should be a multiple of translated_size, e.g. : When we convert each byte of uint8_t type data to rmt format data, the relation between item_num and translated_size should be $item_num = translated_size * 8$.

Parameters

- **src**: Pointer to the buffer storing the raw data that needs to be converted to rmt format.
- **[out] dest**: Pointer to the buffer storing the rmt format data.
- **src_size**: The raw data size.
- **wanted_num**: The number of rmt format data that wanted to get.
- **[out] translated_size**: The size of the raw data that has been converted to rmt format, it should return 0 if no data is converted in user callback.
- **[out] item_num**: The number of the rmt format data that actually converted to, it can be less than wanted_num if there is not enough raw data, but cannot exceed wanted_num. it should return 0 if no data was converted.

Header File

- [soc/include/hal/rmt_types.h](#)

Structures

struct rmt_channel_status_result_t

Data struct of RMT channel status.

Public Members

`rmt_channel_status_t status[RMT_CHANNEL_MAX]`
Store the current status of each channel

Type Definitions

`typedef rmt_channel_id_t rmt_channel_t`
RMT Channel Type.

Enumerations

`enum rmt_mem_owner_t`
RMT Internal Memory Owner.

Values:

`RMT_MEM_OWNER_TX`
RMT RX mode, RMT transmitter owns the memory block

`RMT_MEM_OWNER_RX`
RMT RX mode, RMT receiver owns the memory block

`RMT_MEM_OWNER_MAX`

`enum rmt_source_clk_t`
Clock Source of RMT Channel.

Values:

`RMT_BASECLK_REF`
RMT source clock system reference tick, 1MHz by default (not supported in this version)

`RMT_BASECLK_APB`
RMT source clock is APB CLK, 80Mhz by default

`RMT_BASECLK_MAX`

`enum rmt_data_mode_t`
RMT Data Mode.

Note We highly recommended to use MEM mode not FIFO mode since there will be some gotcha in FIFO mode.

Values:

`RMT_DATA_MODE_FIFO`

`RMT_DATA_MODE_MEM`

`RMT_DATA_MODE_MAX`

`enum rmt_mode_t`
RMT Channel Working Mode (TX or RX)

Values:

`RMT_MODE_TX`
RMT TX mode

`RMT_MODE_RX`
RMT RX mode

`RMT_MODE_MAX`

`enum rmt_idle_level_t`
RMT Idle Level.

Values:

RMT_IDLE_LEVEL_LOW

RMT TX idle level: low Level

RMT_IDLE_LEVEL_HIGH

RMT TX idle level: high Level

RMT_IDLE_LEVEL_MAX

enum rmt_carrier_level_t

RMT Carrier Level.

Values:

RMT_CARRIER_LEVEL_LOW

RMT carrier wave is modulated for low Level output

RMT_CARRIER_LEVEL_HIGH

RMT carrier wave is modulated for high Level output

RMT_CARRIER_LEVEL_MAX

enum rmt_channel_status_t

RMT Channel Status.

Values:

RMT_CHANNEL_UNINIT

RMT channel uninitialized

RMT_CHANNEL_IDLE

RMT channel status idle

RMT_CHANNEL_BUSY

RMT channel status busy

2.3.11 SD Pull-up Requirements

Espressif hardware products are designed for multiple use cases which may require different pull states on pins. For this reason, the pull state of particular pins on certain products will need to be adjusted to provide the pull-ups required in the SD bus.

SD pull-up requirements apply to cases where ESP32 uses the SPI controller to communicate with SD cards. When an SD card is operating in SPI mode or 1-bit SD mode, the CMD and DATA (DAT0 - DAT3) lines of the SD bus must be pulled up by 10 kOhm resistors. Slaves should also have pull-ups on all above-mentioned lines (regardless of whether these lines are connected to the host) in order to prevent SD cards from entering a wrong state.

By default, the MTDI bootstrapping pin is incompatible with the DAT2 line pull-up if the flash voltage is 3.3 V. For more information, see [MTDI Strapping Pin](#) below.

This document has the following structure:

- [Overview of compatibility](#) between the default pull states on pins of Espressif's products and the states required by the SD bus
- [Solutions](#) - ideas on how to resolve compatibility issues
- [Related information](#) - other relevant information

Overview of Compatibility

This section provides an overview of compatibility issues that might occur when using SDIO (secure digital input output). Since the SD bus needs to be connected to pull-ups, these issues should be resolved regardless of whether they are related to master (host) or slave (device). Each issue has links to its respective solution. A solution for a host and device may differ.

Systems on a Chip (SoCs)

- ESP32 (except for D2WD versions, see [ESP32 datasheet](#)):
 - *No Pull-ups*
 - *Conflicts Between Bootstrap and SDIO on DAT2* for models with 3.3 V flash chip
- ESP32-D2WD:
 - *No Pull-ups*
 - *No Pull-up on GPIO12*

Systems in Packages (SIP)

- ESP32-PICO-D4:
 - *No Pull-ups*
 - *Conflicts Between Bootstrap and SDIO on DAT2*

Modules

- ESP32-WROOM-32 Series, including ESP32-WROOM-32, ESP32-WROOM-32D, ESP32-WROOM-32U, and ESP32-SOLO-1
 - *No Pull-ups*
 - *Conflicts Between Bootstrap and SDIO on DAT2*
- ESP32-WROVER Series, including ESP32-WROVER and ESP32-WROVER-I
 - *No Pull-ups*
- ESP32-WROVER-B Series, including ESP32-WROVER-B and ESP32-WROVER-IB
 - *No Pull-ups*
 - *Conflicts Between Bootstrap and SDIO on DAT2*

Development Boards

- ESP32-PICO-KIT, including PICO-KIT v4.1, v4.0, and v3
 - *No Pull-ups*
 - *Conflicts Between Bootstrap and SDIO on DAT2*
 - *Download Mode Not Working (minor issue)*
- ESP32-DevKitC, including ESP32-DevKitC v4 and v2
 - *No Pull-ups*
 - *Conflicts Between Bootstrap and SDIO on DAT2*
 - *Download Mode Not Working (minor issue)*
- ESP-WROVER-KIT
 - Required pull-ups are provided
 - *Pull-up Conflicts on GPIO13* (v4.1, v3, v2, and v1)
 - *Conflicts Between Bootstrap and SDIO on DAT2* (v4.1, v2, and v1)
 - *Download Mode Not Working (minor issue)* (v2, v1)

You can determine the version of your ESP23-WROVER-KIT by checking which module is mounted on it:

- ESP32-WROVER-B on v4.1
 - ESP32-WROVER on v3
 - ESP32-WROOM-32 on v1 and v2
- ESP32-LyraTD-MSD
 - Required pull-ups are provided
 - *Conflicts Between Bootstrap and SDIO on DAT2*
 - ESP32-LyraT
 - Required pull-ups are provided
 - *Pull-up Conflicts on GPIO13*

Non-Espressif Hosts Please make sure that your SDIO host provides necessary pull-ups for all SD bus signals.

Solutions

No Pull-ups If you use a development board without pull-ups, you can do the following:

- If your host and slave device are on separate boards, replace one of them with a board that has pull-ups. For the list of Espressif's development boards with pull-ups, go to [Development Boards](#).
- Attach external pull-ups by connecting each pin which requires a pull-up to VDD via a 10 kOhm resistor.

Pull-up Conflicts on GPIO13 If DAT3 of your device is not properly pulled up, you have the following options:

- Use 1-bit SD mode and tie the device's DAT3 to VDD
- Use SPI mode
- **Perform one of the following actions on the GPIO13 pin:**
 - Remove the pull-down resistors
 - Attach a pull-up resistor of less than 5 kOhm (2 kOhm suggested)
 - Pull it up or drive it high either by using the host or with 3.3 V on VDD in 1-bit SD mode

Conflicts Between Bootstrap and SDIO on DAT2 There is a conflict between the boot strapping requirements of the ESP32 and the SDIO protocol. For details, see [MTDI Strapping Pin](#).

To resolve the conflict, you have the following options:

1. (Recommended) Burn the flash voltage selection eFuses. This will permanently configure the internal regulator's output voltage to 3.3 V, and GPIO12 will not be used as a bootstrapping pin. After that, connect a pull-up resistor to GPIO12.

Warning: Burning eFuses is irreversible! The issue list above might be out of date, so please make sure that the module you are burning has a 3.3 V flash chip by checking the information on <http://www.espressif.com/>. If you burn the 3.3 V eFuses on a module with a 1.8 V flash chip, the module will stop functioning.

If you are sure that you need to irreversibly burn eFuses, go to your ESP-IDF directory and run the following command:

```
components/esptool_py/esptool/espefuse.py set_flash_voltage 3.3V
```

This command will burn the `XPD_SDIO_TIEH`, `XPD_SDIO_FORCE`, and `XPD_SDIO_REG` eFuses. After all the three eFuses are burned to value 1, the internal VDD_SDIO flash voltage regulator will be permanently set to 3.3 V. You will see the following log if the burning succeeds:

```
espefuse.py v2.6
Connecting....

Enable internal flash voltage regulator (VDD_SDIO) to 3.3 V.
The following eFuses are burned: XPD_SDIO_FORCE, XPD_SDIO_REG, XPD_SDIO_TIEH.
This is an irreversible operation.
Type 'BURN' (all capitals) to continue.
BURN
VDD_SDIO setting complete.
```

To check the status of the eFuses, run:

```
``components/esptool_py/esptool/espefuse.py summary``
```

If running from an automated flashing script, `espefuse.py` has an option `--do-not-confirm`. For more details, see the [ESP32 Technical Reference Manual](#) (PDF).

2. **If using 1-bit SD mode or SPI mode**, disconnect the DAT2 pin and make sure it is pulled high. For this, do one the following:
 - Leave the host's DAT2 floating and directly connect the slave's DAT2 to VDD.
 - For a slave device, build a firmware with the option `SDIO_SLAVE_FLAG_DAT2_DISABLED` and re-flash your device. This option will help avoid slave detecting on the DAT2 line. Note that 4-bit SD mode will no longer be supported by the standard Card Common Control Register (CCCR); however, the host will not be aware of that. The use of 4-bit SD mode will have to be disabled on the host's side.

No Pull-up on GPIO12 Your module is compatible with the SDIO protocol. Just connect GPIO12 to VDD via a 10 kOhm resistor.

Download Mode Not Working (minor issue) When the GPIO2 pin is pulled high in accordance with the SD pull-up requirements, you cannot enter Download mode because GPIO2 is a bootstrapping pin which in this case must be pulled low.

There are the following solutions:

- For boards that require shorting the GPIO0 and GPIO2 pins with a jumper, put the jumper in place, and the auto-reset circuit will pull GPIO2 low along with GPIO0 before entering Download mode.
- For boards with components attached to their GPIO2 pin (such as pull-down resistors and/or LEDs), check the schematic of your development board for anything connected to GPIO2.
 - **LEDs** would not affect operation in most cases.
 - **Pull-down resistors** can interfere with DAT0 signals and must be removed.

If the above solutions do not work for you, please determine if it is the host or slave device that has pull-ups affecting their GPIO2, then locate these pull-ups and remove them.

Related Information

MTDI Strapping Pin MTDI (GPIO12) is used as a bootstrapping pin to select the output voltage of an internal regulator (VDD_SDIO) which powers the flash chip. This pin has an internal pull-down, so, if left unconnected, it will read low at startup, which will lead to selecting the default 3.3 V operation.

All ESP32-WROVER modules, excluding ESP32-WROVER-B, use 1.8 V flash and have internal pull-ups on GPIO12. Other modules that use 3.3 V flash have no pull-ups on the GPIO12 pin, and this pin is slightly pulled down internally.

When adding a pull-up to this pin for SD card operation, consider the following:

- For boards that do not use the internal regulator (VDD_SDIO) to power flash, GPIO12 can be pulled high.
- For boards using 1.8 V flash chips, GPIO12 needs to be pulled high at reset. This is fully compatible with the SD card operation.
- On boards using the internal regulator and a 3.3 V flash chip, GPIO12 must be pulled low at reset. This is incompatible with the SD card operation. For reference information on compatibility of Espressif's boards with the SD card operation, see [Overview of Compatibility](#).

Internal Pull-ups and Strapping Requirements Using external resistors is always preferable. However, Espressif's products have internal weak pull-up and pull-down resistors which can be enabled and used instead of external ones. Please keep in mind that this solution cannot guarantee reliable SDIO communication.

With that said, the information about these internal pull-ups and strapping requirements can still be useful. Espressif hardware products have different weak internal pull-ups / pull-downs connected to CMD and DATA pins. The table below shows the default pull-up and pull-down states of the CMD and DATA pins.

The following abbreviations are used in the table:

- **WPU**: Weak pull-up inside the SoC
- **WPD**: Weak pull-down inside the SoC
- **PU**: Pull-up inside Espressif modules but outside the SoC

Table 2: Default pull-up and pull-down states of the CMD and DATA pins

| GPIO number | Pin Name | Startup State | Strapping Requirement |
|-------------|----------|---|---|
| 15 | CMD | WPU | |
| 2 | DAT0 | WPD | Low for Download mode |
| 4 | DAT1 | WPD | |
| 12 | DAT2 | PU for 1.8 V flash; WPD for 3.3 V flash | High for 1.8 V flash; Low for 3.3 V flash |
| 13 | DAT3 | WPU | |

2.3.12 SDMMC Host Driver

Overview

ESP32's SDMMC host peripheral has two slots:

- Slot 0 (`SDMMC_HOST_SLOT_0`) is an 8-bit slot. It uses HS1_* signals in the PIN MUX.
- Slot 1 (`SDMMC_HOST_SLOT_1`) is a 4-bit slot. It uses HS2_* signals in the PIN MUX.

Pin mappings of these slots are given in the table below.

| Signal | Slot 0 | Slot 1 |
|--------|---------------------------|--------|
| CMD | GPIO11 | GPIO15 |
| CLK | GPIO6 | GPIO14 |
| D0 | GPIO7 | GPIO2 |
| D1 | GPIO8 | GPIO4 |
| D2 | GPIO9 | GPIO12 |
| D3 | GPIO10 | GPIO13 |
| D4 | GPIO16 | |
| D5 | GPIO17 | |
| D6 | GPIO5 | |
| D7 | GPIO18 | |
| CD | any input via GPIO matrix | |
| WP | any input via GPIO matrix | |

The Card Detect and Write Protect signals can be routed to arbitrary pins using the GPIO matrix. To reserve the pins, set the `gpio_cd` and `gpio_wp` members of the `sdmmc_slot_config_t` structure before calling `sdmmc_host_init_slot()`. Please note that it is not advised to specify a Card Detect pin when working with SDIO cards, because the card detect signal in ESP32 can also trigger SDIO slave interrupt.

Warning: Pins used by Slot 0 (HS1_*) are also used to connect the SPI flash chip in ESP32-WROOM and ESP32-WROVER modules. These pins cannot be shared between an SD card and SPI flash. If you need to use Slot 0, connect SPI flash to different pins and set eFuses accordingly.

Supported Speed Modes

SDMMC Host driver supports the following speed modes:

- Default Speed (20 MHz), 4-line/1-line (with SD cards), and 8-line (with 3.3 V eMMC)
- High Speed (40 MHz), 4-line/1-line (with SD cards), and 8-line (with 3.3 V eMMC)
- High Speed DDR (40 MHz), 4-line (with 3.3 V eMMC)

Speed modes not supported at present:

- High Speed DDR mode, 8-line eMMC
- UHS-I 1.8 V modes, 4-line SD cards

Using the SDMMC Host Driver

Of all the functions listed below, only the following ones will be used directly by most applications:

- `sdmmc_host_init()`
- `sdmmc_host_init_slot()`
- `sdmmc_host_deinit()`

Other functions, such as the ones given below, will be called by the SD/MMC protocol layer via function pointers in the `sdmmc_host_t` structure:

- `sdmmc_host_set_bus_width()`
- `sdmmc_host_set_card_clk()`
- `sdmmc_host_do_transaction()`

Configuring Bus Width and Frequency

With the default initializers for `sdmmc_host_t` and `sdmmc_slot_config_t` (`SDMMC_HOST_DEFAULT` and `SDMMC_SLOT_CONFIG_DEFAULT`), SDMMC Host driver will attempt to use the widest bus supported by the card (4 lines for SD, 8 lines for eMMC) and the frequency of 20 MHz.

In the designs where communication at 40 MHz frequency can be achieved, it is possible to increase the bus frequency by changing the `max_freq_khz` field of `sdmmc_host_t`:

```
sdmmc_host_t host = SDMMC_HOST_DEFAULT();
host.max_freq_khz = SDMMC_FREQ_HIGHSPEED;
```

To configure the bus width, set the `width` field of `sdmmc_slot_config_t`. For example, to set 1-line mode:

```
sdmmc_slot_config_t slot = SDMMC_SLOT_CONFIG_DEFAULT();
slot.width = 1;
```

See also

See *SD/SDIO/MMC Driver* for the higher level driver which implements the protocol layer.

See *SD SPI Host Driver* for a similar driver which uses the SPI controller and is limited to SD protocol's SPI mode.

See *SD Pull-up Requirements* for pullup support and compatibilities of modules and development kits.

API Reference

Header File

- `driver/include/driver/sdmmc_host.h`

Functions

`esp_err_t sdmmc_host_init` (void)

Initialize SDMMC host peripheral.

Note This function is not thread safe

Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_STATE` if `sdmmc_host_init` was already called
- `ESP_ERR_NO_MEM` if memory can not be allocated

`esp_err_t sdmmc_host_init_slot` (int slot, const `sdmmc_slot_config_t` *slot_config)

Initialize given slot of SDMMC peripheral.

On the ESP32, SDMMC peripheral has two slots:

- Slot 0: 8-bit wide, maps to HS1_* signals in PIN MUX
- Slot 1: 4-bit wide, maps to HS2_* signals in PIN MUX

Card detect and write protect signals can be routed to arbitrary GPIOs using GPIO matrix.

Note This function is not thread safe

Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_STATE` if host has not been initialized using `sdmmc_host_init`

Parameters

- `slot`: slot number (`SDMMC_HOST_SLOT_0` or `SDMMC_HOST_SLOT_1`)

- `slot_config`: additional configuration for the slot

esp_err_t `sdmmc_host_set_bus_width` (int *slot*, size_t *width*)

Select bus width to be used for data transfer.

SD/MMC card must be initialized prior to this command, and a command to set bus width has to be sent to the card (e.g. `SD_APP_SET_BUS_WIDTH`)

Note This function is not thread safe

Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_ARG` if slot number or width is not valid

Parameters

- `slot`: slot number (`SDMMC_HOST_SLOT_0` or `SDMMC_HOST_SLOT_1`)
- `width`: bus width (1, 4, or 8 for slot 0; 1 or 4 for slot 1)

size_t `sdmmc_host_get_slot_width` (int *slot*)

Get bus width configured in `sdmmc_host_init_slot` to be used for data transfer.

Return configured bus width of the specified slot.

Parameters

- `slot`: slot number (`SDMMC_HOST_SLOT_0` or `SDMMC_HOST_SLOT_1`)

esp_err_t `sdmmc_host_set_card_clk` (int *slot*, uint32_t *freq_khz*)

Set card clock frequency.

Currently only integer fractions of 40MHz clock can be used. For High Speed cards, 40MHz can be used. For Default Speed cards, 20MHz can be used.

Note This function is not thread safe

Return

- `ESP_OK` on success
- other error codes may be returned in the future

Parameters

- `slot`: slot number (`SDMMC_HOST_SLOT_0` or `SDMMC_HOST_SLOT_1`)
- `freq_khz`: card clock frequency, in kHz

esp_err_t `sdmmc_host_set_bus_ddr_mode` (int *slot*, bool *ddr_enabled*)

Enable or disable DDR mode of SD interface.

Return

- `ESP_OK` on success
- `ESP_ERR_NOT_SUPPORTED` if DDR mode is not supported on this slot

Parameters

- `slot`: slot number (`SDMMC_HOST_SLOT_0` or `SDMMC_HOST_SLOT_1`)
- `ddr_enabled`: enable or disable DDR mode

esp_err_t `sdmmc_host_do_transaction` (int *slot*, *sdmmc_command_t* **cmdinfo*)

Send command to the card and get response.

This function returns when command is sent and response is received, or data is transferred, or timeout occurs.

Note This function is not thread safe w.r.t. `init/deinit` functions, and bus width/clock speed configuration functions. Multiple tasks can call `sdmmc_host_do_transaction` as long as other `sdmmc_host_*` functions are not called.

Attention Data buffer passed in `cmdinfo->data` must be in DMA capable memory

Return

- `ESP_OK` on success
- `ESP_ERR_TIMEOUT` if response or data transfer has timed out
- `ESP_ERR_INVALID_CRC` if response or data transfer CRC check has failed
- `ESP_ERR_INVALID_RESPONSE` if the card has sent an invalid response
- `ESP_ERR_INVALID_SIZE` if the size of data transfer is not valid in SD protocol
- `ESP_ERR_INVALID_ARG` if the data buffer is not in DMA capable memory

Parameters

- `slot`: slot number (`SDMMC_HOST_SLOT_0` or `SDMMC_HOST_SLOT_1`)

- `cmdinfo`: pointer to structure describing command and data to transfer

esp_err_t **sdmmc_host_io_int_enable** (int *slot*)

Enable IO interrupts.

This function configures the host to accept SDIO interrupts.

Return returns `ESP_OK`, other errors possible in the future

Parameters

- `slot`: slot number (`SDMMC_HOST_SLOT_0` or `SDMMC_HOST_SLOT_1`)

esp_err_t **sdmmc_host_io_int_wait** (int *slot*, TickType_t *timeout_ticks*)

Block until an SDIO interrupt is received, or timeout occurs.

Return

- `ESP_OK` on success (interrupt received)
- `ESP_ERR_TIMEOUT` if the interrupt did not occur within `timeout_ticks`

Parameters

- `slot`: slot number (`SDMMC_HOST_SLOT_0` or `SDMMC_HOST_SLOT_1`)
- `timeout_ticks`: number of RTOS ticks to wait for the interrupt

esp_err_t **sdmmc_host_deinit** (void)

Disable SDMMC host and release allocated resources.

Note This function is not thread safe

Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_STATE` if `sdmmc_host_init` function has not been called

esp_err_t **sdmmc_host_pullup_en** (int *slot*, int *width*)

Enable the pull-ups of sd pins.

Note You should always place actual pullups on the lines instead of using this function. Internal pullup resistance are high and not sufficient, may cause instability in products. This is for debug or examples only.

Return

- `ESP_OK`: if success
- `ESP_ERR_INVALID_ARG`: if configured width larger than maximum the slot can support

Parameters

- `slot`: Slot to use, normally set it to 1.
- `width`: Bit width of your configuration, 1 or 4.

Structures

struct `sdmmc_slot_config_t`

Extra configuration for SDMMC peripheral slot

Public Members

gpio_num_t `gpio_cd`

GPIO number of card detect signal.

gpio_num_t `gpio_wp`

GPIO number of write protect signal.

uint8_t `width`

Bus width used by the slot (might be less than the max width supported)

uint32_t `flags`

Features used by this slot.

Macros

SDMMC_HOST_SLOT_0

SDMMC slot 0.

SDMMC_HOST_SLOT_1

SDMMC slot 1.

SDMMC_HOST_DEFAULT ()

Default *sdmmc_host_t* structure initializer for SDMMC peripheral.

Uses SDMMC peripheral, with 4-bit mode enabled, and max frequency set to 20MHz

SDMMC_SLOT_FLAG_INTERNAL_PULLUP

Enable internal pullups on enabled pins. The internal pullups are insufficient however, please make sure external pullups are connected on the bus. This is for debug / example purpose only.

SDMMC_SLOT_NO_CD

indicates that card detect line is not used

SDMMC_SLOT_NO_WP

indicates that write protect line is not used

SDMMC_SLOT_WIDTH_DEFAULT

use the default width for the slot (8 for slot 0, 4 for slot 1)

SDMMC_SLOT_CONFIG_DEFAULT ()

Macro defining default configuration of SDMMC host slot

2.3.13 SD SPI Host Driver

Overview

The SD SPI host driver allows using the SPI2 (HSPI) or SPI3 (VSPI) controller for communication with SD cards. This driver's naming pattern was adopted from the *SDMMC Host* driver due to their similarity. Likewise, the APIs of both drivers are also very similar.

The SD SPI host driver has the following modes:

- **SPI mode:** offers lower throughput but makes pin selection more flexible. With the help of the GPIO matrix, an SPI peripheral's signals can be routed to any ESP32 pin.
- **1-bit SD mode:** offers higher throughput but requires routing the signals through their dedicated IO_MUX pins only.

The SD SPI driver uses software-controlled CS signal.

Currently, the SD SPI driver cannot handle multi-threaded environments as does not support time-division multiplexing on the same SPI bus. It means that if your application needs to communicate with an SD card and other devices on the same SPI bus, the application itself must ensure that its different tasks do not try to access the SPI slaves at the same time.

How to Use

The state and configurations of the SD SPI host driver are stored in a *sdmmc_host_t* structure. This structure can be initialized using the *SDSPI_HOST_DEFAULT* macro.

The state and configurations of the SD slot are stored in a *sdmmc_slot_config_t* structure. Use the macro *SDSPI_SLOT_CONFIG_DEFAULT* to initialize the structure and to fill in the default pin mappings (SD mode pin mappings).

Only the following driver's API functions are normally used by most applications:

- *sdspi_host_init ()*
- *sdspi_host_init_slot ()*
- *sdspi_host_deinit ()*

Other functions are mostly used by the protocol level SD/SDIO/MMC driver via function pointers in the `sdmmc_host_t` structure. For more details, see *the SD/SDIO/MMC Driver*.

Note: SD over SPI does not support speeds above `SDMMC_FREQ_DEFAULT` due to the limitations of the SPI driver.

API Reference

Header File

- `driver/include/driver/sdspi_host.h`

Functions

`esp_err_t sdspi_host_init` (void)

Initialize SD SPI driver.

Note This function is not thread safe

Return

- ESP_OK on success
- other error codes may be returned in future versions

`esp_err_t sdspi_host_init_slot` (int slot, const `sdspi_slot_config_t` *slot_config)

Initialize SD SPI driver for the specific SPI controller.

Note This function is not thread safe

Note The SDIO over sdspi needs an extra interrupt line. Call `gpio_install_isr_service()` before this function.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if `sdspi_init_slot` has invalid arguments
- ESP_ERR_NO_MEM if memory can not be allocated
- other errors from the underlying `spi_master` and `gpio` drivers

Parameters

- slot: SPI controller to use (HSPI_HOST or VSPI_HOST)
- slot_config: pointer to slot configuration structure

`esp_err_t sdspi_host_do_transaction` (int slot, `sdmmc_command_t` *cmdinfo)

Send command to the card and get response.

This function returns when command is sent and response is received, or data is transferred, or timeout occurs.

Note This function is not thread safe w.r.t. `init/deinit` functions, and bus width/clock speed configuration functions. Multiple tasks can call `sdspi_host_do_transaction` as long as other `sdspi_host_*` functions are not called.

Return

- ESP_OK on success
- ESP_ERR_TIMEOUT if response or data transfer has timed out
- ESP_ERR_INVALID_CRC if response or data transfer CRC check has failed
- ESP_ERR_INVALID_RESPONSE if the card has sent an invalid response

Parameters

- slot: SPI controller (HSPI_HOST or VSPI_HOST)
- cmdinfo: pointer to structure describing command and data to transfer

`esp_err_t sdspi_host_set_card_clk` (int slot, uint32_t freq_khz)

Set card clock frequency.

Currently only integer fractions of 40MHz clock can be used. For High Speed cards, 40MHz can be used. For Default Speed cards, 20MHz can be used.

Note This function is not thread safe

Return

- ESP_OK on success
- other error codes may be returned in the future

Parameters

- `slot`: SPI controller (HSPI_HOST or VSPI_HOST)
- `freq_khz`: card clock frequency, in kHz

esp_err_t **sdspi_host_deinit** (void)

Release resources allocated using `sdspi_host_init`.

Note This function is not thread safe

Return

- ESP_OK on success
- ESP_ERR_INVALID_STATE if `sdspi_host_init` function has not been called

esp_err_t **sdspi_host_io_int_enable** (int *slot*)

Enable SDIO interrupt.

Return

- ESP_OK on success

Parameters

- `slot`: SPI controller to use (HSPI_HOST or VSPI_HOST)

esp_err_t **sdspi_host_io_int_wait** (int *slot*, TickType_t *timeout_ticks*)

Wait for SDIO interrupt until timeout.

Return

- ESP_OK on success

Parameters

- `slot`: SPI controller to use (HSPI_HOST or VSPI_HOST)
- `timeout_ticks`: Ticks to wait before timeout.

Structures

struct **sdspi_slot_config_t**

Extra configuration for SPI host

Public Members

gpio_num_t **gpio_miso**

GPIO number of MISO signal.

gpio_num_t **gpio_mosi**

GPIO number of MOSI signal.

gpio_num_t **gpio_sck**

GPIO number of SCK signal.

gpio_num_t **gpio_cs**

GPIO number of CS signal.

gpio_num_t **gpio_cd**

GPIO number of card detect signal.

gpio_num_t **gpio_wp**

GPIO number of write protect signal.

gpio_num_t **gpio_int**

GPIO number of interrupt line (input) for SDIO card.

int **dma_channel**

DMA channel to be used by SPI driver (1 or 2)

Macros**SDSPI_HOST_DEFAULT ()**

Default `sdmmc_host_t` structure initializer for SD over SPI driver.

Uses SPI mode and max frequency set to 20MHz

‘slot’ can be set to one of HSPI_HOST, VSPI_HOST.

SDSPI_SLOT_NO_CD

indicates that card detect line is not used

SDSPI_SLOT_NO_WP

indicates that write protect line is not used

SDSPI_SLOT_NO_INT

indicates that interrupt line is not used

SDSPI_SLOT_CONFIG_DEFAULT ()

Macro defining default configuration of SPI host

2.3.14 SDIO Card Slave Driver**Overview**

The ESP32 SDIO Card peripherals (Host, Slave) shares two sets of pins as below table. The first set is usually occupied by SPI0 bus which is responsible for the SPI flash holding the code to run. This means SDIO slave driver can only runs on the second set of pins while SDIO host is not using it.

The SDIO slave can run under 3 modes: SPI, 1-bit SD and 4-bit SD modes, which is detected automatically by the hardware. According to the SDIO specification, CMD and DAT0-3 lines should be pulled up no matter in 1-bit, 4-bit or SPI mode.

Connections

| Pin Name | Corresponding pins in SPI mode | Slot1 | Slot2 |
|----------|--------------------------------|-------------|-------|
| | | GPIO Number | |
| CLK | SCLK | 6 | 14 |
| CMD | MOSI | 11 | 15 |
| DAT0 | MISO | 7 | 2 |
| DAT1 | Interrupt | 8 | 4 |
| DAT2 | N.C. (pullup) | 9 | 12 |
| DAT3 | #CS | 10 | 13 |

- 1-bit SD mode: Connect CLK, CMD, DAT0, DAT1 pins and the ground.
- 4-bit SD mode: Connect all pins and the ground.
- SPI mode: Connect SCLK, MOSI, MISO, Interrupt, #CS pins and the ground.

Note: Please check if CMD and DATA lines D0-D3 of the card are properly pulled up by 10 KOhm resistors. This should be ensured even in 1-bit mode or SPI mode. Most official modules don't offer these pullups internally. If you are using official development boards, check [Overview of Compatibility](#) to see whether your development boards have such pullups.

Note: Most official modules have conflicts on strapping pins with the SDIO slave function. If you are using a ESP32 module with 3.3 V flash inside, you have to burn the EFUSE when you are developing on the module for the first time. See [Overview of Compatibility](#) to see how to make your modules compatible with the SDIO.

Here is a list for modules/kits with 3.3 V flash:

- Modules: ESP32-PICO-D4, ESP32-WROOM-32 series (including ESP32-SOLO-1), ESP32-WROVER-B and ESP32-WROVER-IB

- Kits: ESP32-PICO-KIT, ESP32-DevKitC (till v4), ESP32-WROVER-KIT (v4.1 (also known as ESP32-WROVER-KIT-VB), v2, v1 (also known as DevKitJ v1))

You can tell the version of your ESP32-WROVER-KIT version from the module on it: v4.1 are with ESP32-WROVER-B modules, v3 are with ESP32-WROVER modules, while v2 and v1 are with ESP32-WROOM-32 modules.

Refer to [SD Pull-up Requirements](#) for more technical details of the pullups.

The host initialize the slave into SD mode by first sending CMD0 with DAT3 pin high, or in SPI mode by sending CMD0 with CS pin (the same pin as DAT3) low.

After the initialization, the host can enable the 4-bit SD mode by writing CCCR register 0x07 by CMD52. All the bus detection process are handled by the slave peripheral.

The host has to communicate with the slave by an ESP-slave-specific protocol. The slave driver offers 3 services over Function 1 access by CMD52 and CMD53: (1) a sending FIFO and a receiving FIFO, (2) 52 8-bit R/W registers shared by host and slave, (3) 16 interrupt sources (8 from host to slave, and 8 from slave to host).

Terminology The SDIO slave driver uses the following terms:

- Transfer: a transfer is always started by a command token from the host, and may contain a reply and several data blocks. ESP32 slave software is based on transfers.
- Sending: slave to host transfers.
- Receiving: host to slave transfers.

Note: Register names in ESP Rechnical Reference Manual are oriented from the point of view of the host, i.e. ‘rx’ registers refer to sending, while ‘tx’ registers refer to receiving. We’re not using *tx* or *rx* in the driver to avoid ambiguities.

- FIFO: specific address in Function 1 that can be access by CMD53 to read/write large amount of data. The address is related to the length requested to read from/write to the slave in a single transfer: *requested length* = 0x1F800-address.
- Ownership: When the driver takes ownership of a buffer, it means the driver can randomly read/write the buffer (usually via DMA). The application should not read/write the buffer until the ownership is returned to the application. If the application reads from a buffer owned by a receiving driver, the data read can be random; if the application writes to a buffer owned by a sending driver, the data sent may be corrupted.
- Requested length: The length requested in one transfer determined by the FIFO address.
- Transfer length: The length requested in one transfer determined by the CMD53 byte/block count field.

Note: Requested length is different from the transfer length. ESP32 slave DMA base on the *requested length* rather than the *transfer length*. The *transfer length* should be no shorter than the *requested length*, and the rest part will be filled with 0 (sending) or discard (receiving).

- Receiving buffer size: The buffer size is pre-defined between the host and the slave before communication starts. Slave application has to set the buffer size during initialization by the `recv_buffer_size` member of `sdio_slave_config_t`.
- Interrupts: the esp32 slave support interrupts in two directions: from host to slave (called slave interrupts below) and from slave to host (called host interrupts below). See more in [Interrupts](#).
- Registers: specific address in Function 1 access by CMD52 or CMD53.

Communication with ESP SDIO Slave The host should initialize the ESP32 SDIO slave according to the standard SDIO initialization process (Sector 3.1.2 of [SDIO Simplified Specification](#)), which is described briefly in [ESP SDIO slave initialization](#).

Furthermore, there's an ESP32-specific upper-level communication protocol upon the CMD52/CMD53 to Func 1. Please refer to [ESP SDIO slave protocol](#). There is also a component [ESP Serial Slave Link](#) for ESP32 master to communicate with ESP32 SDIO slave, see example [peripherals/sdio](#) when programming your host.

Communication with ESP SDIO Slave

ESP SDIO slave initialization The host should initialize the ESP32 SDIO slave according to the standard SDIO initialization process (Sector 3.1.2 of [SDIO Simplified Specification](#)). In this specification and below, the SDIO slave is also called an (SD)IO card. All the initialization CMD52 and CMD53 are sent to Func 0 (CIA region). Here is an brief example on how to do this:

1. **SDIO reset** CMD52 (Write 0x6=0x8)
2. **SD reset** CMD0
3. **Check whether IO card (optional)** CMD8
4. **Send SDIO op cond and wait for card ready** CMD5 arg = 0x00000000
 CMD5 arg = 0x00ff8000 (according to the response above, poll until ready)
Example: Arg of R4 after first CMD5 (arg=0x00000000) is 0xXXFFFF00.
 Keep sending CMD5 with arg=0x00FFFF00 until the R4 shows card ready (arg bit 31=1).
5. **Set address** CMD3
6. **Select card** CMD7 (arg address according to CMD3 response)
Example: Arg of R6 after CMD3 is 0x0001xxxx.
 Arg of CMD7 should be 0x00010000.
7. **Select 4-bit mode (optional)** CMD52 (Write 0x07=0x02)
8. **Enable func1** CMD52 (Write 0x02=0x02)
9. **Enable SDIO interrupt (required if interrupt line (DAT1) is used)** CMD52 (Write 0x04=0x03)
10. **Set Func0 blocksize (optional, default value is 512 (0x200))** CMD52/53 (Read 0x10~0x11)
 CMD52/53 (Write 0x10=0x00)
 CMD52/53 (Write 0x11=0x02)
 CMD52/53 (Read 0x10~0x11, read to check the final value)
11. **Set Func1 blocksize (optional, default value is 512 (0x200))** CMD52/53 (Read 0x110~0x111)
 CMD52/53 (Write 0x110=0x00)
 CMD52/53 (Write 0x111=0x02)
 CMD52/53 (Read 0x110~0x111, read to check the final value)

ESP SDIO slave protocol The protocol is based on Function 1 access by CMD52 and CMD53, offering 3 services: (1) sending and receiving FIFO, (2) 52 8-bit R/W register shared by host and slave, (3) 8 general purpose interrupt sources from host to slave and 8 in the opposite direction.

There is a component [esp_serial_slave_link](#) implementing the logic of this protocol for ESP32 master to communicate with the ESP32 slave. See [ESP Serial Slave Link](#).

The host should access the registers below as described to communicate with slave.

Slave register table

32-bit

- 0x044 (TOKEN_RDATA): in which bit 27-16 holds the receiving buffer number.
- 0x058 (INT_ST): holds the interrupt source bits from slave to host.
- 0x060 (PKT_LEN): holds the accumulated length (by byte) to be sent from slave to host.
- 0x0D4 (INT_CLR): write 1 to clear interrupt bits corresponding to INT_ST.
- 0x0DC (INT_ENA): mask bits for interrupts from slave to host.

8-bit Shared general purpose registers:

- 0x06C-0x077: R/W registers 0-11 shared by slave and host.

- 0x07A-0x07B: R/W registers 14-15 shared by slave and host.
- 0x07E-0x07F: R/W registers 18-19 shared by slave and host.
- 0x088-0x08B: R/W registers 24-27 shared by slave and host.
- 0x09C-0x0BB: R/W registers 32-63 shared by slave and host.

Interrupt Registers: - 0x08D (SLAVE_INT): bits for host to interrupt slave. auto clear.

FIFO (sending and receiving) 0x090 - 0x1F7FF are reserved for FIFOs.

The address of CMD53 is related to the length requested to read from/write to the slave in a single transfer:

$$\text{requested length} = 0x1F800 - \text{address}$$

The slave will respond with the length according to the length field in CMD53, with the data longer than *requested length* filled with 0 (sending) or discard (receiving).

Note: This includes both the block and the byte mode of CMD53.

The function number should be set to 1, OP Code should be set to 1 (for CMD53).

It is allowed to use CMD53 mode combination of block+byte to get higher efficiency when accessing the FIFO by arbitrary length. E.g. The block size is set to 512 by default, you can write/get 1031 bytes of data to/from the FIFO by:

1. Send CMD53 in block mode, block count=2 (1024 bytes) to address 0x1F3F9=0x1F800-**1031**.
 2. Then send CMD53 in byte mode, byte count=8 (or 7 if your controller supports that) to address 0x1F7F9=0x1F800-7.
-

Interrupts For the host interrupts, the slave raise the interrupt by pulling DAT1 line down at a proper time (level sensitive). The host detect this and read the INT_ST register to see the source. Then the host can clear it by writing the INT_CLR register and do something with the interrupt. The host can also mask unneeded sources by clearing the bits in INT_ENA register corresponding to the sources. If all the sources are cleared (or masked), the DAT1 line goes inactive.

sdio_slave_hostint_t (*SDIO Card Slave Driver*) shows the bit definition corresponding to host interrupt sources.

For the slave interrupts, the host send transfers to write the SLAVE_INT register. Once a bit is written from 0 to 1, the slave hardware and driver will detect it and inform the app.

Receiving FIFO To write the receiving FIFO in the slave, host should work in the following steps:

1. Read the TOKEN1 field (bits 27-16) of TOKEN_RDATA (0x044) register. The buffer number remaining is TOKEN1 minus the number of buffers used by host.
2. Make sure the buffer number is sufficient ($\text{recv_buffer_size} * \text{buffer_num}$ is greater than data to write, *recv_buffer_size* is pre-defined between the host and the slave before the communication starts). Or go back to step 1 until the buffer is enough.
3. Write to the FIFO address with CMD53. Note that the *requested length* should not be larger than calculated in step 2, and the FIFO address is related to *requested length*.
4. Calculate used buffers, note that non-full buffer at the tail should be seen as one that is used.

Sending FIFO To read the sending FIFO in the slave, host should work in the following steps:

1. Wait for the interrupt line to be active (optional, low by default).
2. Read (poll) the interrupt bits in INT_ST register to see whether new packets exists.
3. If new packets are ready, reads the PKT_LEN reg. The data length to read from slave is PKT_LEN minuses the length that has been read from the host. If the PKT_LEN is not larger than used, wait and poll until the slave is ready and update the PKT_LEN.

4. Read from the FIFO with CMD53. Note that the *requested length* should not be larger than calculated in step3, and the FIFO address is related to *requested length*.
5. Recorded read length.

Interrupts There are interrupts from host to slave, and from slave to host to help communicating conveniently.

Slave Interrupts The host can interrupt the slave by writing any one bit in the register 0x08D. Once any bit of the register is set, an interrupt is raised and the SDIO slave driver calls the callback function defined in the `slave_intr_cb` member in the `sdio_slave_config_t` structure.

Note: The callback function is called in the ISR, do not use any delay, loop or spinlock in the callback.

There's another set of functions can be used. You can call `sdio_slave_wait_int` to wait for an interrupt within a certain time, or call `sdio_slave_clear_int` to clear interrupts from host. The callback function can work with the wait functions perfectly.

Host Interrupts The slave can interrupt the host by an interrupt line (at certain time) which is level sensitive. When the host see the interrupt line pulled down, it may read the slave interrupt status register, to see the interrupt source. Host can clear interrupt bits, or choose to disable a interrupt source. The interrupt line will hold active until all the sources are cleared or disabled.

There are several dedicated interrupt sources as well as general purpose sources. see `sdio_slave_hostint_t` for more information.

Shared Registers There are 52 8-bit R/W shared registers to share information between host and slave. The slave can write or read the registers at any time by `sdio_slave_read_reg` and `sdio_slave_write_reg`. The host can access (R/W) the register by CMD52 or CMD53.

Receiving FIFO When the host is going to send the slave some packets, it has to check whether the slave is ready to receive by reading the buffer number of slave.

To allow the host sending data to the slave, the application has to load buffers to the slave driver by the following steps:

1. Register the buffer by calling `sdio_slave_recv_register_buf`, and get the handle of the registered buffer. The driver will allocate memory for the linked-list descriptor needed to link the buffer onto the hardware.
2. Load buffers onto the driver by passing the buffer handle to `sdio_slave_recv_load_buf`.
3. Call `sdio_slave_recv` to get the received data. If non-blocking call is needed, set `wait=0`.
4. Pass the handle of processed buffer back to the driver by `sdio_recv_load_buf` again.

Note: To avoid overhead from copying data, the driver itself doesn't have any buffer inside, the application is responsible to offer new buffers in time. The DMA will automatically store received data to the buffer.

Sending FIFO Each time the slave has data to send, it raises an interrupt and the host will request for the packet length. There are two sending modes:

- Stream Mode: when a buffer is loaded to the driver, the buffer length will be counted into the packet length requested by host in the incoming communications. Regardless previous packets are sent or not. This means the host can get data of several buffers in one transfer.
- Packet Mode: the packet length is updated packet by packet, and only when previous packet is sent. This means that the host can only get data of one buffer in one transfer.

Note: To avoid overhead from copying data, the driver itself doesn't have any buffer inside. Namely, the DMA takes data directly from the buffer provided by the application. The application should not touch the buffer until the sending is finished.

The sending mode can be set in the `sending_mode` member of `sdio_slave_config_t`, and the buffer numbers can be set in the `send_queue_size`. All the buffers are restricted to be no larger than 4092 bytes. Though in the stream mode several buffers can be sent in one transfer, each buffer is still counted as one in the queue.

The application can call `sdio_slave_transmit` to send packets. In this case the function returns when the transfer is successfully done, so the queue is not fully used. When higher efficiency is required, the application can use the following functions instead:

1. Pass buffer information (address, length, as well as an `arg` indicating the buffer) to `sdio_slave_send_queue`. If non-blocking call is needed, set `wait=0`. If the `wait` is not `portMAX_DELAY` (wait until success), application has to check the result to know whether the data is put in to the queue or discard.
2. Call `sdio_slave_send_get_finished` to get and deal with a finished transfer. A buffer should be keep unmodified until returned from `sdio_slave_send_get_finished`. This means the buffer is actually sent to the host, rather than just staying in the queue.

There are several ways to use the `arg` in the queue parameter:

1. Directly point `arg` to a dynamic-allocated buffer, and use the `arg` to free it when transfer finished.
2. Wrap transfer informations in a transfer structure, and point `arg` to the structure. You can use the structure to do more things like:

```
typedef struct {
    uint8_t* buffer;
    size_t   size;
    int      id;
}sdio_transfer_t;

//and send as:
sdio_transfer_t trans = {
    .buffer = ADDRESS_TO_SEND,
    .size = 8,
    .id = 3, //the 3rd transfer so far
};
sdio_slave_send_queue(trans.buffer, trans.size, &trans, portMAX_DELAY);

//... maybe more transfers are sent here

//and deal with finished transfer as:
sdio_transfer_t* arg = NULL;
sdio_slave_send_get_finished((void**)&arg, portMAX_DELAY);
ESP_LOGI("tag", "(%d) successfully send %d bytes of %p", arg->id, arg->size,
    ↪arg->buffer);
some_post_callback(arg); //do more things
```

3. Working with the receiving part of this driver, point `arg` to the receive buffer handle of this buffer. So that we can directly use the buffer to receive data when it's sent:

```
uint8_t buffer[256]={1,2,3,4,5,6,7,8};
sdio_slave_buf_handle_t handle = sdio_slave_rcv_register_buf(buffer);
sdio_slave_send_queue(buffer, 8, handle, portMAX_DELAY);

//... maybe more transfers are sent here

//and load finished buffer to receive as
sdio_slave_buf_handle_t handle = NULL;
sdio_slave_send_get_finished((void**)&handle, portMAX_DELAY);
sdio_slave_rcv_load_buf(handle);
```

More about this, see [peripherals/sdio](#).

Application Example

Slave/master communication: [peripherals/sdio](#).

API Reference

Header File

- [soc/include/hal/sdio_slave_types.h](#)

Enumerations

enum sdio_slave_hostint_t

Mask of interrupts sending to the host.

Values:

SDIO_SLAVE_HOSTINT_BIT0 = BIT(0)

General purpose interrupt bit 0.

SDIO_SLAVE_HOSTINT_BIT1 = BIT(1)

SDIO_SLAVE_HOSTINT_BIT2 = BIT(2)

SDIO_SLAVE_HOSTINT_BIT3 = BIT(3)

SDIO_SLAVE_HOSTINT_BIT4 = BIT(4)

SDIO_SLAVE_HOSTINT_BIT5 = BIT(5)

SDIO_SLAVE_HOSTINT_BIT6 = BIT(6)

SDIO_SLAVE_HOSTINT_BIT7 = BIT(7)

SDIO_SLAVE_HOSTINT_SEND_NEW_PACKET = BIT(23)

New packet available.

enum sdio_slave_timing_t

Timing of SDIO slave.

Values:

SDIO_SLAVE_TIMING_PSEND_PSAMPLE = 0

Send at posedge, and sample at posedge. Default value for HS mode. Normally there's no problem using this to work in DS mode.

SDIO_SLAVE_TIMING_NSEND_PSAMPLE

Send at negedge, and sample at posedge. Default value for DS mode and below.

SDIO_SLAVE_TIMING_PSEND_NSAMPLE

Send at posedge, and sample at negedge.

SDIO_SLAVE_TIMING_NSEND_NSAMPLE

Send at negedge, and sample at negedge.

enum sdio_slave_sending_mode_t

Configuration of SDIO slave mode.

Values:

SDIO_SLAVE_SEND_STREAM = 0

Stream mode, all packets to send will be combined as one if possible.

SDIO_SLAVE_SEND_PACKET = 1

Packet mode, one packets will be sent one after another (only increase packet_len if last packet sent).

Header File

- `driver/include/driver/sdio_slave.h`

Functions

esp_err_t **sdio_slave_initialize** (*sdio_slave_config_t* **config*)

Initialize the sdio slave driver

Return

- `ESP_ERR_NOT_FOUND` if no free interrupt found.
- `ESP_ERR_INVALID_STATE` if already initialized.
- `ESP_ERR_NO_MEM` if fail due to memory allocation failed.
- `ESP_OK` if success

Parameters

- *config*: Configuration of the sdio slave driver.

void **sdio_slave_deinit** (void)

De-initialize the sdio slave driver to release the resources.

esp_err_t **sdio_slave_start** (void)

Start hardware for sending and receiving, as well as set the `IOREADY1` to 1.

Note The driver will continue sending from previous data and `PKT_LEN` counting, keep data received as well as start receiving from current `TOKEN1` counting. See `sdio_slave_reset`.

Return

- `ESP_ERR_INVALID_STATE` if already started.
- `ESP_OK` otherwise.

void **sdio_slave_stop** (void)

Stop hardware from sending and receiving, also set `IOREADY1` to 0.

Note this will not clear the data already in the driver, and also not reset the `PKT_LEN` and `TOKEN1` counting. Call `sdio_slave_reset` to do that.

esp_err_t **sdio_slave_reset** (void)

Clear the data still in the driver, as well as reset the `PKT_LEN` and `TOKEN1` counting.

Return always return `ESP_OK`.

sdio_slave_buf_handle_t **sdio_slave_recv_register_buf** (uint8_t **start*)

Register buffer used for receiving. All buffers should be registered before used, and then can be used (again) in the driver by the handle returned.

Note The driver will use and only use the amount of space specified in the `recv_buffer_size` member set in the `sdio_slave_config_t`. All buffers should be larger than that. The buffer is used by the DMA, so it should be DMA capable and 32-bit aligned.

Return The buffer handle if success, otherwise `NULL`.

Parameters

- *start*: The start address of the buffer.

esp_err_t **sdio_slave_recv_unregister_buf** (*sdio_slave_buf_handle_t* *handle*)

Unregister buffer from driver, and free the space used by the descriptor pointing to the buffer.

Return `ESP_OK` if success, `ESP_ERR_INVALID_ARG` if the handle is `NULL` or the buffer is being used.

Parameters

- *handle*: Handle to the buffer to release.

esp_err_t **sdio_slave_recv_load_buf** (*sdio_slave_buf_handle_t* *handle*)

Load buffer to the queue waiting to receive data. The driver takes ownership of the buffer until the buffer is returned by `sdio_slave_send_get_finished` after the transaction is finished.

Return

- `ESP_ERR_INVALID_ARG` if invalid handle or the buffer is already in the queue. Only after the buffer is returned by `sdio_slave_recv` can you load it again.
- `ESP_OK` if success

Parameters

- `handle`: Handle to the buffer ready to receive data.

`esp_err_t sdio_slave_recv (sdio_slave_buf_handle_t *handle_ret, uint8_t **out_addr, size_t *out_len, TickType_t wait)`

Get received data if exist. The driver returns the ownership of the buffer to the app.

Note Call `sdio_slave_load_buf` with the handle to re-load the buffer onto the link list, and receive with the same buffer again. The address and length of the buffer got here is the same as got from `sdio_slave_get_buffer`.

Return

- `ESP_ERR_INVALID_ARG` if `handle_ret` is NULL
- `ESP_ERR_TIMEOUT` if timeout before receiving new data
- `ESP_OK` if success

Parameters

- `handle_ret`: Handle to the buffer holding received data. Use this handle in `sdio_slave_recv_load_buf` to receive in the same buffer again.
- [out] `out_addr`: Output of the start address, set to NULL if not needed.
- [out] `out_len`: Actual length of the data in the buffer, set to NULL if not needed.
- `wait`: Time to wait before data received.

`uint8_t *sdio_slave_recv_get_buf (sdio_slave_buf_handle_t handle, size_t *len_o)`

Retrieve the buffer corresponding to a handle.

Return buffer address if success, otherwise NULL.

Parameters

- `handle`: Handle to get the buffer.
- `len_o`: Output of buffer length

`esp_err_t sdio_slave_send_queue (uint8_t *addr, size_t len, void *arg, TickType_t wait)`

Put a new sending transfer into the send queue. The driver takes ownership of the buffer until the buffer is returned by `sdio_slave_send_get_finished` after the transaction is finished.

Return

- `ESP_ERR_INVALID_ARG` if the length is not greater than 0.
- `ESP_ERR_TIMEOUT` if the queue is still full until timeout.
- `ESP_OK` if success.

Parameters

- `addr`: Address for data to be sent. The buffer should be DMA capable and 32-bit aligned.
- `len`: Length of the data, should not be longer than 4092 bytes (may support longer in the future).
- `arg`: Argument to returned in `sdio_slave_send_get_finished`. The argument can be used to indicate which transaction is done, or as a parameter for a callback. Set to NULL if not needed.
- `wait`: Time to wait if the buffer is full.

`esp_err_t sdio_slave_send_get_finished (void **out_arg, TickType_t wait)`

Return the ownership of a finished transaction.

Return `ESP_ERR_TIMEOUT` if no transaction finished, or `ESP_OK` if succeed.

Parameters

- `out_arg`: Argument of the finished transaction. Set to NULL if unused.
- `wait`: Time to wait if there's no finished sending transaction.

`esp_err_t sdio_slave_transmit (uint8_t *addr, size_t len)`

Start a new sending transfer, and wait for it (blocked) to be finished.

Return

- `ESP_ERR_INVALID_ARG` if the length of descriptor is not greater than 0.
- `ESP_ERR_TIMEOUT` if the queue is full or host do not start a transfer before timeout.
- `ESP_OK` if success.

Parameters

- `addr`: Start address of the buffer to send
- `len`: Length of buffer to send.

`uint8_t sdio_slave_read_reg` (*int pos*)

Read the spi slave register shared with host.

Note register 28 to 31 are reserved for interrupt vector.

Return value of the register.

Parameters

- *pos*: register address, 0-27 or 32-63.

esp_err_t `sdio_slave_write_reg` (*int pos*, *uint8_t reg*)

Write the spi slave register shared with host.

Note register 29 and 31 are used for interrupt vector.

Return `ESP_ERR_INVALID_ARG` if address wrong, otherwise `ESP_OK`.

Parameters

- *pos*: register address, 0-11, 14-15, 18-19, 24-27 and 32-63, other address are reserved.
- *reg*: the value to write.

sdio_slave_hostint_t `sdio_slave_get_host_intena` (*void*)

Get the interrupt enable for host.

Return the interrupt mask.

void `sdio_slave_set_host_intena` (*sdio_slave_hostint_t mask*)

Set the interrupt enable for host.

Parameters

- *mask*: Enable mask for host interrupt.

esp_err_t `sdio_slave_send_host_int` (*uint8_t pos*)

Interrupt the host by general purpose interrupt.

Return

- `ESP_ERR_INVALID_ARG` if interrupt num error
- `ESP_OK` otherwise

Parameters

- *pos*: Interrupt num, 0-7.

void `sdio_slave_clear_host_int` (*sdio_slave_hostint_t mask*)

Clear general purpose interrupt to host.

Parameters

- *mask*: Interrupt bits to clear, by bit mask.

esp_err_t `sdio_slave_wait_int` (*int pos*, *TickType_t wait*)

Wait for general purpose interrupt from host.

Note this clears the interrupt at the same time.

Return `ESP_OK` if success, `ESP_ERR_TIMEOUT` if timeout.

Parameters

- *pos*: Interrupt source number to wait for. is set.
- *wait*: Time to wait before interrupt triggered.

Structures

`struct sdio_slave_config_t`

Configuration of SDIO slave.

Public Members

sdio_slave_timing_t `timing`

timing of `sdio_slave`. see `sdio_slave_timing_t`.

sdio_slave_sending_mode_t `sending_mode`

mode of `sdio_slave`. `SDIO_SLAVE_MODE_STREAM` if the data needs to be sent as much as possible; `SDIO_SLAVE_MODE_PACKET` if the data should be sent in packets.

int **send_queue_size**

max buffers that can be queued before sending.

size_t **recv_buffer_size**

If `buffer_size` is too small, it costs more CPU time to handle larger number of buffers. If `buffer_size` is too large, the space larger than the transaction length is left blank but still counts a buffer, and the buffers are easily run out. Should be set according to length of data really transferred. All data that do not fully fill a buffer is still counted as one buffer. E.g. 10 bytes data costs 2 buffers if the size is 8 bytes per buffer. Buffer size of the slave pre-defined between host and slave before communication. All receive buffer given to the driver should be larger than this.

sdio_event_cb_t **event_cb**

when the host interrupts slave, this callback will be called with interrupt number (0-7).

uint32_t **flags**

Features to be enabled for the slave, combinations of `SDIO_SLAVE_FLAG_*`.

Macros

SDIO_SLAVE_RECV_MAX_BUFFER

SDIO_SLAVE_FLAG_DAT2_DISABLED

It is required by the SD specification that all 4 data lines should be used and pulled up even in 1-bit mode or SPI mode. However, as a feature, the user can specify this flag to make use of DAT2 pin in 1-bit mode. Note that the host cannot read CCCR registers to know we don't support 4-bit mode anymore, please do this at your own risk.

SDIO_SLAVE_FLAG_HOST_INTR_DISABLED

The DAT1 line is used as the interrupt line in SDIO protocol. However, as a feature, the user can specify this flag to make use of DAT1 pin of the slave in 1-bit mode. Note that the host has to do polling to the interrupt registers to know whether there are interrupts from the slave. And it cannot read CCCR registers to know we don't support 4-bit mode anymore, please do this at your own risk.

SDIO_SLAVE_FLAG_INTERNAL_PULLUP

Enable internal pullups for enabled pins. It is required by the SD specification that all the 4 data lines should be pulled up even in 1-bit mode or SPI mode. Note that the internal pull-ups are not sufficient for stable communication, please do connect external pull-ups on the bus. This is only for example and debug use.

Type Definitions

typedef void (**sdio_event_cb_t*) (uint8_t event)

typedef void **sdio_slave_buf_handle_t*

Handle of a receive buffer, register a handle by calling `sdio_slave_recv_register_buf`. Use the handle to load the buffer to the driver, or call `sdio_slave_recv_unregister_buf` if it is no longer used.

2.3.15 Sigma-delta Modulation

Introduction

ESP32 has a second-order sigma-delta modulation module. This driver configures the channels of the sigma-delta module.

Functionality Overview

There are eight independent sigma-delta modulation channels identified with *sigmadelta_channel_t*. Each channel is capable to output the binary, hardware generated signal with the sigma-delta modulation.

Selected channel should be set up by providing configuration parameters in *sigmadelta_config_t* and then applying this configuration with *sigmadelta_config()*.

Another option is to call individual functions, that will configure all required parameters one by one:

- **Prescaler** of the sigma-delta generator - `sigmadelta_set_prescale()`
- **Duty** of the output signal - `sigmadelta_set_duty()`
- **GPIO pin** to output modulated signal - `sigmadelta_set_pin()`

The range of the 'duty' input parameter of `sigmadelta_set_duty()` is from -128 to 127 (eight bit signed integer). If zero value is set, then the output signal's duty will be about 50%, see description of `sigmadelta_set_duty()`.

Application Example

Sigma-delta Modulation example: [peripherals/sigmadelta](#).

API Reference

Header File

- [driver/include/driver/sigmadelta.h](#)

Functions

`esp_err_t sigmadelta_config(const sigmadelta_config_t *config)`

Configure Sigma-delta channel.

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE sigmadelta driver already initialized
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `config`: Pointer of Sigma-delta channel configuration struct

`esp_err_t sigmadelta_set_duty(sigmadelta_channel_t channel, int8_t duty)`

Set Sigma-delta channel duty.

This function is used to set Sigma-delta channel duty, If you add a capacitor between the output pin and ground, the average output voltage will be $V_{dc} = V_{DDIO} / 256 * duty + V_{DDIO}/2$, where VDDIO is the power supply voltage.

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE sigmadelta driver has not been initialized
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `channel`: Sigma-delta channel number
- `duty`: Sigma-delta duty of one channel, the value ranges from -128 to 127, recommended range is -90 ~ 90. The waveform is more like a random one in this range.

`esp_err_t sigmadelta_set_prescale(sigmadelta_channel_t channel, uint8_t prescale)`

Set Sigma-delta channel's clock pre-scale value. The source clock is APP_CLK, 80MHz. The clock frequency of the sigma-delta channel is APP_CLK / pre_scale.

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE sigmadelta driver has not been initialized
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `channel`: Sigma-delta channel number
- `prescale`: The divider of source clock, ranges from 0 to 255

`esp_err_t sigmadelta_set_pin(sigmadelta_channel_t channel, gpio_num_t gpio_num)`

Set Sigma-delta signal output pin.

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE sigmadelta driver has not been initialized
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `channel`: Sigma-delta channel number
- `gpio_num`: GPIO number of output pin.

Header File

- [soc/include/hal/sigmadelta_types.h](#)

Structures**struct sigmadelta_config_t**

Sigma-delta configure struct.

Public Members*sigmadelta_channel_t* **channel**

Sigma-delta channel number

int8_t sigmadelta_duty

Sigma-delta duty, duty ranges from -128 to 127.

uint8_t sigmadelta_prescale

Sigma-delta prescale, prescale ranges from 0 to 255.

uint8_t sigmadelta_gpio

Sigma-delta output io number, refer to gpio.h for more details.

Type Definitions**typedef int sigmadelta_port_t**

SIGMADELTA port number, the max port number is (SIGMADELTA_NUM_MAX -1).

typedef int sigmadelta_channel_t

Sigma-delta channel list.

2.3.16 SPI Master Driver

SPI Master driver is a program that controls ESP32's SPI peripherals while they function as masters.

Overview of ESP32's SPI peripherals

ESP32 integrates four SPI peripherals.

- SPI0 and SPI1 are used internally to access the ESP32's attached flash memory and thus are currently not open to users. They share one signal bus via an arbiter.
- SPI2 and SPI3 are general purpose SPI controllers, sometimes referred to as HSPI and VSPI, respectively. They are open to users. SPI2 and SPI3 have independent signal buses with the same respective names. Each bus has three CS lines to drive up to three SPI slaves.

Terminology

The terms used in relation to the SPI master driver are given in the table below.

| Term | Definition |
|--------------------|--|
| Host | The SPI controller peripheral inside ESP32 that initiates SPI transmissions over the bus, and acts as an SPI Master. This may be the SPI2 or SPI3 peripheral. (The driver will also support the SPI1 peripheral in the future.) |
| Device | SPI slave device. An SPI bus may be connected to one or more Devices. Each Device shares the MOSI, MISO and SCLK signals but is only active on the bus when the Host asserts the Device's individual CS line. |
| Bus | A signal bus, common to all Devices connected to one Host. In general, a bus includes the following lines: MISO, MOSI, SCLK, one or more CS lines, and, optionally, QUADWP and QUADHD. So Devices are connected to the same lines, with the exception that each Device has its own CS line. Several Devices can also share one CS line if connected in the daisy-chain manner. |
| • MISO | Master In, Slave Out, a.k.a. Q. Data transmission from a Device to Host. |
| • MOSI | Master Out, Slave In, a.k.a. D. Data transmission from a Host to Device. |
| • SCLK | Serial Clock. Oscillating signal generated by a Host that keeps the transmission of data bits in sync. |
| • CS | Chip Select. Allows a Host to select individual Device(s) connected to the bus in order to send or receive data. |
| • QUADWP | Write Protect signal. Only used for 4-bit (qio/qout) transactions. |
| • QUADHD | Hold signal. Only used for 4-bit (qio/qout) transactions. |
| • Assertion | The action of activating a line. The opposite action of returning the line back to inactive (back to idle) is called <i>de-assertion</i> . |
| Transaction | One instance of a Host asserting a CS line, transferring data to and from a Device, and de-asserting the CS line. Transactions are atomic, which means they can never be interrupted by another transaction. |
| Launch edge | Edge of the clock at which the source register <i>launches</i> the signal onto the line. |
| Latch edge | Edge of the clock at which the destination register <i>latches in</i> the signal. |

Driver Features

The SPI master driver governs communications of Hosts with Devices. The driver supports the following features:

- Multi-threaded environments
- Transparent handling of DMA transfers while reading and writing data
- Automatic time-division multiplexing of data coming from different Devices on the same signal bus

Warning: The SPI master driver has the concept of multiple Devices connected to a single bus (sharing a single ESP32 SPI peripheral). As long as each Device is accessed by only one task, the driver is thread safe. However, if multiple tasks try to access the same SPI Device, the driver is **not thread-safe**. In this case, it is recommended to either:

- Refactor your application so that each SPI peripheral is only accessed by a single task at a time.
- Add a mutex lock around the shared Device using `xSemaphoreCreateMutex`.

SPI Transactions

An SPI bus transaction consists of five phases which can be found in the table below. Any of these phases can be skipped.

| Phase | Description |
|----------------|--|
| Command | In this phase, a command (0-16 bit) is written to the bus by the Host. |
| Address | In this phase, an address (0-64 bit) is transmitted over the bus by the Host. |
| Write | Host sends data to a Device. This data follows the optional command and address phases and is indistinguishable from them at the electrical level. |
| Dummy | This phase is configurable and is used to meet the timing requirements. |
| Read | Device sends data to its Host. |

The attributes of a transaction are determined by the bus configuration structure `spi_bus_config_t`, device configuration structure `spi_device_interface_config_t`, and transaction configuration structure `spi_transaction_t`.

An SPI Host can send full-duplex transactions, during which the read and write phases occur simultaneously. The total transaction length is determined by the sum of the following members:

- `spi_device_interface_config_t::command_bits`
- `spi_device_interface_config_t::address_bits`
- `spi_transaction_t::length`

While the member `spi_transaction_t::rxlength` only determines the length of data received into the buffer.

In half-duplex transactions, the read and write phases are not simultaneous (one direction at a time). The lengths of the write and read phases are determined by `length` and `rxlength` members of the struct `spi_transaction_t` respectively.

The command and address phases are optional, as not every SPI device requires a command and/or address. This is reflected in the Device's configuration: if `command_bits` and/or `address_bits` are set to zero, no command or address phase will occur.

The read and write phases can also be optional, as not every transaction requires both writing and reading data. If `rx_buffer` is NULL and `SPI_TRANS_USE_RXDATA` is not set, the read phase is skipped. If `tx_buffer` is NULL and `SPI_TRANS_USE_TXDATA` is not set, the write phase is skipped.

The driver supports two types of transactions: the interrupt transactions and polling transactions. The programmer can choose to use a different transaction type per Device. If your Device requires both transaction types, see *Notes on Sending Mixed Transactions to the Same Device*.

Interrupt Transactions Interrupt transactions will block the transaction routine until the transaction completes, thus allowing the CPU to run other tasks.

An application task can queue multiple transactions, and the driver will automatically handle them one-by-one in the interrupt service routine (ISR). It allows the task to switch to other procedures until all the transactions complete.

Polling Transactions Polling transactions do not use interrupts. The routine keeps polling the SPI Host's status bit until the transaction is finished.

All the tasks that use interrupt transactions can be blocked by the queue. At this point, they will need to wait for the ISR to run twice before the transaction is finished. Polling transactions save time otherwise spent on queue handling and context switching, which results in smaller transaction intervals. The disadvantage is that the CPU is busy while these transactions are in progress.

The `spi_device_polling_end()` routine needs an overhead of at least 1 us to unblock other tasks when the transaction is finished. It is strongly recommended to wrap a series of polling transactions using the functions `spi_device_acquire_bus()` and `spi_device_release_bus()` to avoid the overhead. For more information, see *Bus Acquiring*.

Command and Address Phases During the command and address phases, the members `cmd` and `addr` in the struct `spi_transaction_t` are sent to the bus, nothing is read at this time. The default lengths of the command and address phases are set in `spi_device_interface_config_t` by calling `spi_bus_add_device()`. If the flags `SPI_TRANS_VARIABLE_CMD` and `SPI_TRANS_VARIABLE_ADDR` in the member `spi_transaction_t::flags` are not set, the driver automatically sets the length of these phases to default values during Device initialization.

If the lengths of the command and address phases need to be variable, declare the struct `spi_transaction_ext_t`, set the flags `SPI_TRANS_VARIABLE_CMD` and/or `SPI_TRANS_VARIABLE_ADDR` in the member `spi_transaction_ext_t::base` and configure the rest of base as usual. Then the length of each phase will be equal to `command_bits` and `address_bits` set in the struct `spi_transaction_ext_t`.

Write and Read Phases Normally, the data that needs to be transferred to or from a Device will be read from or written to a chunk of memory indicated by the members `rx_buffer` and `tx_buffer` of the structure `spi_transaction_t`. If DMA is enabled for transfers, the buffers are required to be:

1. Allocated in DMA-capable internal memory. If *external PSRAM is enabled*, this means using `pvPortMallocCaps(size, MALLOC_CAP_DMA)`.
2. 32-bit aligned (starting from a 32-bit boundary and having a length of multiples of 4 bytes).

If these requirements are not satisfied, the transaction efficiency will be affected due to the allocation and copying of temporary buffers.

Note: Half-duplex transactions with both read and write phases are not supported when using DMA. For details and workarounds, see *Known Issues*.

Bus Acquiring Sometimes you might want to send SPI transactions exclusively and continuously so that it takes as little time as possible. For this, you can use bus acquiring, which helps to suspend transactions (both polling or interrupt) to other devices until the bus is released. To acquire and release a bus, use the functions `spi_device_acquire_bus()` and `spi_device_release_bus()`.

Driver Usage

- Initialize an SPI bus by calling the function `spi_bus_initialize()`. Make sure to set the correct I/O pins in the struct `spi_bus_config_t`. Set the signals that are not needed to `-1`.
- Register a Device connected to the bus with the driver by calling the function `spi_bus_add_device()`. Make sure to configure any timing requirements the device might need with the parameter `dev_config`. You should now have obtained the Device's handle which will be used when sending a transaction to it.
- To interact with the Device, fill one or more `spi_transaction_t` structs with any transaction parameters required. Then send the structs either using a polling transaction or an interrupt transaction:

- **Interrupt** Either queue all transactions by calling the function `spi_device_queue_trans()` and, at a later time, query the result using the function `spi_device_get_trans_result()`, or handle all requests synchronously by feeding them into `spi_device_transmit()`.
- **Polling** Call the function `spi_device_polling_transmit()` to send polling transactions. Alternatively, if you want to insert something in between, send the transactions by using `spi_device_polling_start()` and `spi_device_polling_end()`.
- (Optional) To perform back-to-back transactions with a Device, call the function `spi_device_acquire_bus()` before sending transactions and `spi_device_release_bus()` after the transactions have been sent.
- (Optional) To unload the driver for a certain Device, call `spi_bus_remove_device()` with the Device handle as an argument.
- (Optional) To remove the driver for a bus, make sure no more drivers are attached and call `spi_bus_free()`.

The example code for the SPI master driver can be found in the [peripherals/spi_master](#) directory of ESP-IDF examples.

Transactions with Data Not Exceeding 32 Bits When the transaction data size is equal to or less than 32 bits, it will be sub-optimal to allocate a buffer for the data. The data can be directly stored in the transaction struct instead. For transmitted data, it can be achieved by using the `tx_data` member and setting the `SPI_TRANS_USE_TXDATA` flag on the transmission. For received data, use `rx_data` and set `SPI_TRANS_USE_RXDATA`. In both cases, do not touch the `tx_buffer` or `rx_buffer` members, because they use the same memory locations as `tx_data` and `rx_data`.

Transactions with Integers Other Than `uint8_t` An SPI Host reads and writes data into memory byte by byte. By default, data is sent with the most significant bit (MSB) first, as LSB first used in rare cases. If a value less than 8 bits needs to be sent, the bits should be written into memory in the MSB first manner.

For example, if `0b00010` needs to be sent, it should be written into a `uint8_t` variable, and the length for reading should be set to 5 bits. The Device will still receive 8 bits with 3 additional “random” bits, so the reading must be performed correctly.

On top of that, ESP32 is a little-endian chip, which means that the least significant byte of `uint16_t` and `uint32_t` variables is stored at the smallest address. Hence, if `uint16_t` is stored in memory, bits [7:0] are sent first, followed by bits [15:8].

For cases when the data to be transmitted has the size differing from `uint8_t` arrays, the following macros can be used to transform data to the format that can be sent by the SPI driver directly:

- `SPI_SWAP_DATA_TX` for data to be transmitted
- `SPI_SWAP_DATA_RX` for data received

Notes on Sending Mixed Transactions to the Same Device To reduce coding complexity, send only one type of transactions (interrupt or polling) to one Device. However, you still can send both interrupt and polling transactions alternately. The notes below explain how to do this.

The polling transactions should be initiated only after all the polling and interrupt transactions are finished.

Since an unfinished polling transaction blocks other transactions, please do not forget to call the function `spi_device_polling_end()` after `spi_device_polling_start()` to allow other transactions or to allow other Devices to use the bus. Remember that if there is no need to switch to other tasks during your polling transaction, you can initiate a transaction with `spi_device_polling_transmit()` so that it will be ended automatically.

In-flight polling transactions are disturbed by the ISR operation to accommodate interrupt transactions. Always make sure that all the interrupt transactions sent to the ISR are finished before you call `spi_device_polling_start()`. To do that, you can keep calling `spi_device_get_trans_result()` until all the transactions are returned.

To have better control of the calling sequence of functions, send mixed transactions to the same Device only within a single task.

GPIO Matrix and IO_MUX

Most of ESP32's peripheral signals have direct connection to their dedicated IO_MUX pins. However, the signals can also be routed to any other available pins using the less direct GPIO matrix. If at least one signal is routed through the GPIO matrix, then all signals will be routed through it.

The GPIO matrix introduces flexibility of routing but also brings the following disadvantages:

- Increases the input delay of the MISO signal, which makes MISO setup time violations more likely. If SPI needs to operate at high speeds, use dedicated IO_MUX pins.
- Allows signals with clock frequencies only up to 40 MHz, as opposed to 80 MHz if IO_MUX pins are used.

Note: For more details about the influence of the MISO input delay on the maximum clock frequency, see [Timing Considerations](#).

The IO_MUX pins for SPI buses are given below.

| Pin Name | SPI2 | SPI3 |
|----------|-------------|------|
| | GPIO Number | |
| CS0* | 15 | 5 |
| SCLK | 14 | 18 |
| MISO | 12 | 19 |
| MOSI | 13 | 23 |
| QUADWP | 2 | 22 |
| QUADHD | 4 | 21 |

- Only the first Device attached to the bus can use the CS0 pin.

Transfer Speed Considerations

There are three factors limiting the transfer speed:

- Transaction interval
- SPI clock frequency
- Cache miss of SPI functions, including callbacks

The main parameter that determines the transfer speed for large transactions is clock frequency. For multiple small transactions, the transfer speed is mostly determined by the length of transaction intervals.

Transaction Interval Transaction interval is the time that software requires to set up SPI peripheral registers and to copy data to FIFOs, or to set up DMA links.

Interrupt transactions allow appending extra overhead to accommodate the cost of FreeRTOS queues and the time needed for switching between tasks and the ISR.

For **interrupt transactions**, the CPU can switch to other tasks when a transaction is in progress. This saves the CPU time but increases the interval. See [Interrupt Transactions](#). For **polling transactions**, it does not block the task but allows to do polling when the transaction is in progress. For more information, see [Polling Transactions](#).

If DMA is enabled, setting up the linked list requires about 2 us per transaction. When a master is transferring data, it automatically reads the data from the linked list. If DMA is not enabled, the CPU has to write and read each byte from the FIFO by itself. Usually, this is faster than 2 us, but the transaction length is limited to 64 bytes for both write and read.

Typical transaction interval timings for one byte of data are given below.

| | Typical Transaction Time (us) | |
|--------|-------------------------------|---------|
| | Interrupt | Polling |
| DMA | 24 | 8 |
| No DMA | 22 | 7 |

SPI Clock Frequency Transferring each byte takes eight times the clock period $8/f_{spi}$. If the clock frequency is too high, the use of some functions might be limited. See [Timing Considerations](#).

Cache Miss The default config puts only the ISR into the IRAM. Other SPI related functions, including the driver itself and the callback, might suffer from the cache miss and will need to wait until the code is read from the flash. Select [CONFIG_SPI_MASTER_IN_IRAM](#) to put the whole SPI driver into IRAM and put the entire callback(s) and its callee functions into IRAM to prevent cache miss.

For an interrupt transaction, the overall cost is $20+8n/F_{spi}[MHz]$ [us] for n bytes transferred in one transaction. Hence, the transferring speed is: $n/(20+8n/F_{spi})$. An example of transferring speed at 8 MHz clock speed is given in the following table.

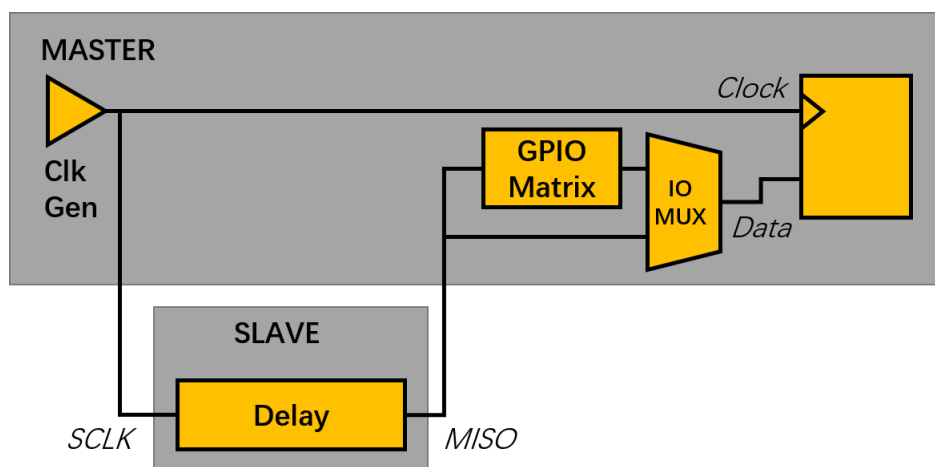
| Frequency (MHz) | Transaction Interval (us) | Transaction Length (bytes) | Total Time (us) | Total Speed (KBps) |
|-----------------|---------------------------|----------------------------|-----------------|--------------------|
| 8 | 25 | 1 | 26 | 38.5 |
| 8 | 25 | 8 | 33 | 242.4 |
| 8 | 25 | 16 | 41 | 490.2 |
| 8 | 25 | 64 | 89 | 719.1 |
| 8 | 25 | 128 | 153 | 836.6 |

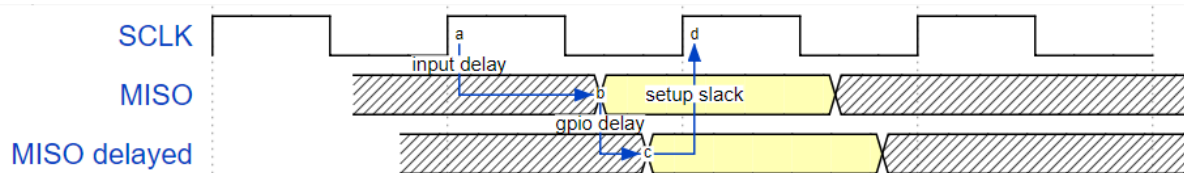
When a transaction length is short, the cost of transaction interval is high. If possible, try to squash several short transactions into one transaction to achieve a higher transfer speed.

Please note that the ISR is disabled during flash operation by default. To keep sending transactions during flash operations, enable [CONFIG_SPI_MASTER_ISR_IN_IRAM](#) and set `ESP_INTR_FLAG_IRAM` in the member `spi_bus_config_t::intr_flags`. In this case, all the transactions queued before starting flash operations will be handled by the ISR in parallel. Also note that the callback of each Device and their callee functions should be in IRAM, or your callback will crash due to cache miss. For more details, see [IRAM-Safe Interrupt Handlers](#).

Timing Considerations

As shown in the figure below, there is a delay on the MISO line after the SCLK launch edge and before the signal is latched by the internal register. As a result, the MISO pin setup time is the limiting factor for the SPI clock speed. When the delay is too long, the setup slack is < 0 , and the setup timing requirement is violated, which results in the failure to perform the reading correctly.





The maximum allowed frequency is dependent on:

- `input_delay_ns` - maximum data valid time on the MISO bus after a clock cycle on SCLK starts
- If the `IO_MUX` pin or the GPIO Matrix is used

When the GPIO matrix is used, the maximum allowed frequency is reduced to about 33~77% in comparison to the existing *input delay*. To retain a higher frequency, you have to use the `IO_MUX` pins or the *dummy bit workaround*. You can obtain the maximum reading frequency of the master by using the function `spi_get_freq_limit()`.

Dummy bit workaround: Dummy clocks, during which the Host does not read data, can be inserted before the read phase begins. The Device still sees the dummy clocks and sends out data, but the Host does not read until the read phase comes. This compensates for the lack of the MISO setup time required by the Host and allows the Host to do reading at a higher frequency.

In the ideal case, if the Device is so fast that the input delay is shorter than an APB clock cycle - 12.5 ns - the maximum frequency at which the Host can read (or read and write) in different conditions is as follows:

| Frequency Limit (MHz) | | Dummy Bits Used By Driver | Comments |
|-----------------------|-------------|---------------------------|-----------------------------|
| GPIO matrix | IO_MUX pins | | |
| 26.6 | 80 | No | |
| 40 | – | Yes | Half-duplex, no DMA allowed |

If the Host only writes data, the *dummy bit workaround* and the frequency check can be disabled by setting the bit `SPI_DEVICE_NO_DUMMY` in the member `spi_device_interface_config_t::flags`. When disabled, the output frequency can be 80MHz, even if the GPIO matrix is used.

`spi_device_interface_config_t::flags`

The SPI master driver can work even if the `input_delay_ns` in the structure `spi_device_interface_config_t` is set to 0. However, setting an accurate value helps to:

- Calculate the frequency limit for full-duplex transactions
- Compensate the timing correctly with dummy bits for half-duplex transactions

You can approximate the maximum data valid time after the launch edge of SPI clocks by checking the statistics in the AC characteristics chapter of your Device's specification or measure the time on an oscilloscope or logic analyzer.

Please note that the actual PCB layout design and the excessive loads may increase the input delay. It means that non-optimal wiring and/or a load capacitor on the bus will most likely lead to the input delay values exceeding the values given in the Device specification or measured while the bus is floating.

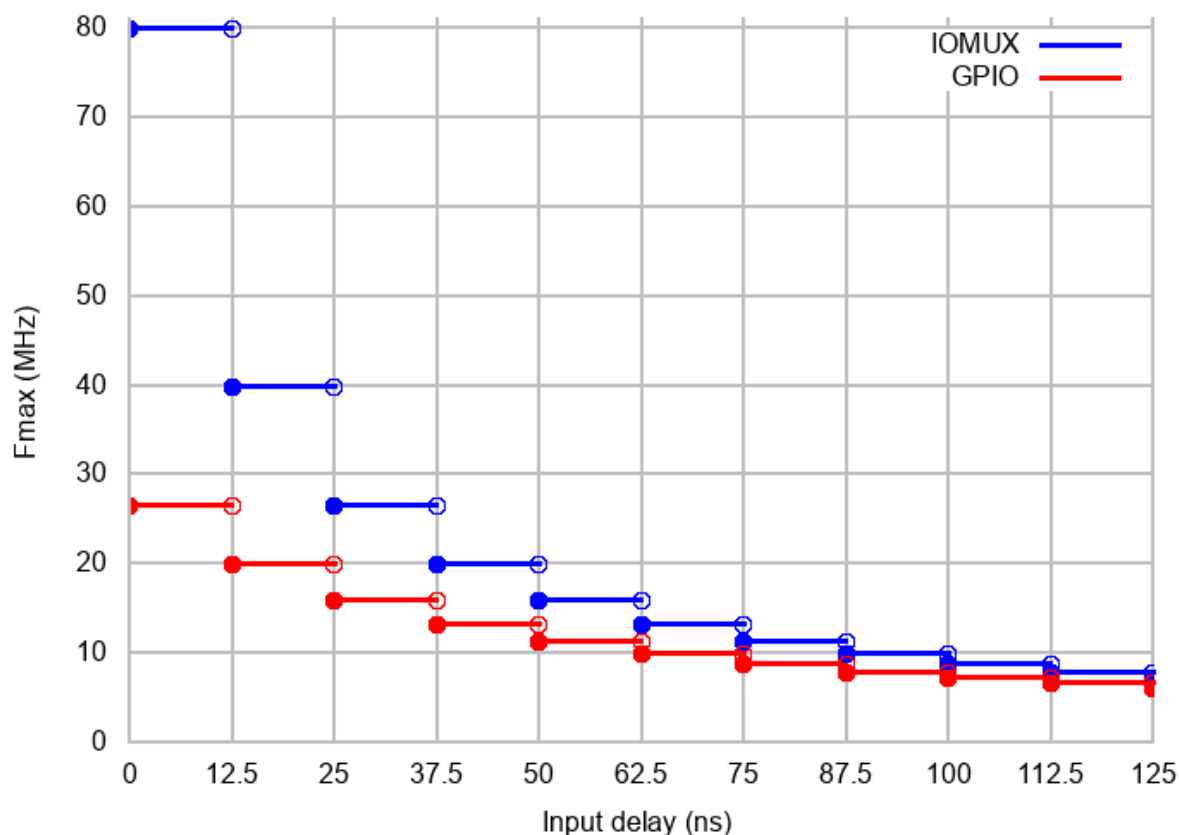
Some typical delay values are shown in the following table.

| Device | Input delay (ns) |
|---|------------------|
| Ideal Device | 0 |
| ESP32 slave using <code>IO_MUX</code> * | 50 |
| ESP32 slave using <code>GPIO_MUX</code> * | 75 |
| ESP32's slave device is on a different physical chip. | |

The MISO path delay (valid time) consists of a slave's *input delay* plus master's *GPIO matrix delay*. This delay determines the frequency limit above which full-duplex transfers will not work as well as the dummy bits used in the half-duplex transactions. The frequency limit is:

$$Freq\ limit\ [MHz] = 80 / (\text{floor}(MISO\ delay[ns]/12.5) + 1)$$

The figure below shows the relationship between frequency limit and input delay. Two extra APB clock cycle periods should be added to the MISO delay if the master uses the GPIO matrix.



Corresponding frequency limits for different Devices with different *input delay* times are shown in the table below.

| Master | Input delay (ns) | MISO path delay (ns) | Freq. limit (MHz) |
|--------------|------------------|----------------------|-------------------|
| IO_MUX (0ns) | 0 | 0 | 80 |
| | 50 | 50 | 16 |
| | 75 | 75 | 11.43 |
| GPIO (25ns) | 0 | 25 | 26.67 |
| | 50 | 75 | 11.43 |
| | 75 | 100 | 8.89 |

Known Issues

- Half-duplex transactions are not compatible with DMA when both writing and reading phases are used. If such transactions are required, you have to use one of the alternative solutions:
 - Use full-duplex transactions instead.
 - Disable DMA by setting the bus initialization function's last parameter to 0 as follows:

```
ret=spi_bus_initialize(VSPI_HOST, &buscfg, 0);
```

This can prohibit you from transmitting and receiving data longer than 64 bytes.
 - Try using the command and address fields to replace the write phase.
- Full-duplex transactions are not compatible with the *dummy bit workaround*, hence the frequency is limited. See [dummy bit speed-up workaround](#).
- `cs_ena_pretrans` is not compatible with the command and address phases of full-duplex transactions.

Application Example

The code example for displaying graphics on an ESP32-WROVER-KIT's 320x240 LCD screen can be found in the [peripherals/spi_master](#) directory of ESP-IDF examples.

API Reference - SPI Common

Header File

- [soc/include/hal/spi_types.h](#)

Enumerations

`enum spi_host_device_t`

Enum with the three SPI peripherals that are software-accessible in it.

Values:

`SPI1_HOST = 0`
SPI1.

`SPI2_HOST = 1`
SPI2.

`SPI3_HOST = 2`
SPI3.

Header File

- [driver/include/driver/spi_common.h](#)

Functions

`esp_err_t spi_bus_initialize` (*spi_host_device_t* host, **const** *spi_bus_config_t* *bus_config, int dma_chan)

Initialize a SPI bus.

Warning For now, only supports HSPI and VSPI.

Warning If a DMA channel is selected, any transmit and receive buffer used should be allocated in DMA-capable memory.

Warning The ISR of SPI is always executed on the core which calls this function. Never starve the ISR on this core or the SPI transactions will not be handled.

Return

- `ESP_ERR_INVALID_ARG` if configuration is invalid
- `ESP_ERR_INVALID_STATE` if host already is in use
- `ESP_ERR_NO_MEM` if out of memory
- `ESP_OK` on success

Parameters

- `host`: SPI peripheral that controls this bus
- `bus_config`: Pointer to a *spi_bus_config_t* struct specifying how the host should be initialized
- `dma_chan`: Either channel 1 or 2, or 0 in the case when no DMA is required. Selecting a DMA channel for a SPI bus allows transfers on the bus to have sizes only limited by the amount of internal memory. Selecting no DMA channel (by passing the value 0) limits the amount of bytes transferred to a maximum of 64. Set to 0 if only the SPI flash uses this bus.

`esp_err_t spi_bus_free` (*spi_host_device_t* host)

Free a SPI bus.

Warning In order for this to succeed, all devices have to be removed first.

Return

- `ESP_ERR_INVALID_ARG` if parameter is invalid
- `ESP_ERR_INVALID_STATE` if not all devices on the bus are freed

- ESP_OK on success

Parameters

- `host`: SPI peripheral to free

Structures

struct spi_bus_config_t

This is a configuration structure for a SPI bus.

You can use this structure to specify the GPIO pins of the bus. Normally, the driver will use the GPIO matrix to route the signals. An exception is made when all signals either can be routed through the IO_MUX or are -1. In that case, the IO_MUX is used, allowing for >40MHz speeds.

Note Be advised that the slave driver does not use the quadwp/quadhd lines and fields in `spi_bus_config_t` referring to these lines will be ignored and can thus safely be left uninitialized.

Public Members

int mosi_io_num

GPIO pin for Master Out Slave In (=spi_d) signal, or -1 if not used.

int miso_io_num

GPIO pin for Master In Slave Out (=spi_q) signal, or -1 if not used.

int sclk_io_num

GPIO pin for Spi CLoCK signal, or -1 if not used.

int quadwp_io_num

GPIO pin for WP (Write Protect) signal which is used as D2 in 4-bit communication modes, or -1 if not used.

int quadhd_io_num

GPIO pin for HD (HoLD) signal which is used as D3 in 4-bit communication modes, or -1 if not used.

int max_transfer_sz

Maximum transfer size, in bytes. Defaults to 4092 if 0 when DMA enabled, or to SOC_SPI_MAXIMUM_BUFFER_SIZE if DMA is disabled.

uint32_t flags

Abilities of bus to be checked by the driver. Or-ed value of SPICOMMON_BUSFLAG_* flags.

int intr_flags

Interrupt flag for the bus to set the priority, and IRAM attribute, see `esp_intr_alloc.h`. Note that the EDGE, INTRDISABLED attribute are ignored by the driver. Note that if ESP_INTR_FLAG_IRAM is set, ALL the callbacks of the driver, and their callee functions, should be put in the IRAM.

Macros

SPI_MAX_DMA_LEN

SPI_SWAP_DATA_TX (DATA, LEN)

Transform unsigned integer of length <= 32 bits to the format which can be sent by the SPI driver directly.

E.g. to send 9 bits of data, you can:

```
uint16_t data = SPI_SWAP_DATA_TX(0x145, 9);
```

Then points `tx_buffer` to `&data`.

Parameters

- `DATA`: Data to be sent, can be `uint8_t`, `uint16_t` or `uint32_t`.
- `LEN`: Length of data to be sent, since the SPI peripheral sends from the MSB, this helps to shift the data to the MSB.

SPI_SWAP_DATA_RX (DATA, LEN)

Transform received data of length ≤ 32 bits to the format of an unsigned integer.

E.g. to transform the data of 15 bits placed in a 4-byte array to integer:

```
uint16_t data = SPI_SWAP_DATA_RX(*(uint32_t*)t->rx_data, 15);
```

Parameters

- DATA: Data to be rearranged, can be `uint8_t`, `uint16_t` or `uint32_t`.
- LEN: Length of data received, since the SPI peripheral writes from the MSB, this helps to shift the data to the LSB.

SPICOMMON_BUSFLAG_SLAVE

Initialize I/O in slave mode.

SPICOMMON_BUSFLAG_MASTER

Initialize I/O in master mode.

SPICOMMON_BUSFLAG_IOMUX_PINS

Check using iomux pins. Or indicates the pins are configured through the IO mux rather than GPIO matrix.

SPICOMMON_BUSFLAG_SCLK

Check existing of SCLK pin. Or indicates CLK line initialized.

SPICOMMON_BUSFLAG_MISO

Check existing of MISO pin. Or indicates MISO line initialized.

SPICOMMON_BUSFLAG_MOSI

Check existing of MOSI pin. Or indicates CLK line initialized.

SPICOMMON_BUSFLAG_DUAL

Check MOSI and MISO pins can output. Or indicates bus able to work under DIO mode.

SPICOMMON_BUSFLAG_WPHD

Check existing of WP and HD pins. Or indicates WP & HD pins initialized.

SPICOMMON_BUSFLAG_QUAD

Check existing of MOSI/MISO/WP/HD pins as output. Or indicates bus able to work under QIO mode.

SPICOMMON_BUSFLAG_NATIVE_PINS**API Reference - SPI Master****Header File**

- [driver/include/driver/spi_master.h](#)

Functions

`esp_err_t spi_bus_add_device` (*spi_host_device_t* host, **const** *spi_device_interface_config_t* *dev_config, *spi_device_handle_t* *handle)

Allocate a device on a SPI bus.

This initializes the internal structures for a device, plus allocates a CS pin on the indicated SPI master peripheral and routes it to the indicated GPIO. All SPI master devices have three CS pins and can thus control up to three devices.

Note While in general, speeds up to 80MHz on the dedicated SPI pins and 40MHz on GPIO-matrix-routed pins are supported, full-duplex transfers routed over the GPIO matrix only support speeds up to 26MHz.

Return

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_NOT_FOUND if host doesn't have any free CS slots
- ESP_ERR_NO_MEM if out of memory
- ESP_OK on success

Parameters

- `host`: SPI peripheral to allocate device on
- `dev_config`: SPI interface protocol config for the device
- `handle`: Pointer to variable to hold the device handle

esp_err_t **spi_bus_remove_device** (*spi_device_handle_t* handle)

Remove a device from the SPI bus.

Return

- `ESP_ERR_INVALID_ARG` if parameter is invalid
- `ESP_ERR_INVALID_STATE` if device already is freed
- `ESP_OK` on success

Parameters

- `handle`: Device handle to free

esp_err_t **spi_device_queue_trans** (*spi_device_handle_t* handle, *spi_transaction_t* *trans_desc, TickType_t ticks_to_wait)

Queue a SPI transaction for interrupt transaction execution. Get the result by `spi_device_get_trans_result`.

Note Normally a device cannot start (queue) polling and interrupt transactions simultaneously.

Return

- `ESP_ERR_INVALID_ARG` if parameter is invalid
- `ESP_ERR_TIMEOUT` if there was no room in the queue before `ticks_to_wait` expired
- `ESP_ERR_NO_MEM` if allocating DMA-capable temporary buffer failed
- `ESP_ERR_INVALID_STATE` if previous transactions are not finished
- `ESP_OK` on success

Parameters

- `handle`: Device handle obtained using `spi_host_add_dev`
- `trans_desc`: Description of transaction to execute
- `ticks_to_wait`: Ticks to wait until there's room in the queue; use `portMAX_DELAY` to never time out.

esp_err_t **spi_device_get_trans_result** (*spi_device_handle_t* handle, *spi_transaction_t* **trans_desc, TickType_t ticks_to_wait)

Get the result of a SPI transaction queued earlier by `spi_device_queue_trans`.

This routine will wait until a transaction to the given device successfully completed. It will then return the description of the completed transaction so software can inspect the result and e.g. free the memory or re-use the buffers.

Return

- `ESP_ERR_INVALID_ARG` if parameter is invalid
- `ESP_ERR_TIMEOUT` if there was no completed transaction before `ticks_to_wait` expired
- `ESP_OK` on success

Parameters

- `handle`: Device handle obtained using `spi_host_add_dev`
- `trans_desc`: Pointer to variable able to contain a pointer to the description of the transaction that is executed. The descriptor should not be modified until the descriptor is returned by `spi_device_get_trans_result`.
- `ticks_to_wait`: Ticks to wait until there's a returned item; use `portMAX_DELAY` to never time out.

esp_err_t **spi_device_transmit** (*spi_device_handle_t* handle, *spi_transaction_t* *trans_desc)

Send a SPI transaction, wait for it to complete, and return the result.

This function is the equivalent of calling `spi_device_queue_trans()` followed by `spi_device_get_trans_result()`. Do not use this when there is still a transaction separately queued (started) from `spi_device_queue_trans()` or `polling_start/transmit` that hasn't been finalized.

Note This function is not thread safe when multiple tasks access the same SPI device. Normally a device cannot start (queue) polling and interrupt transactions simultaneously.

Return

- `ESP_ERR_INVALID_ARG` if parameter is invalid
- `ESP_OK` on success

Parameters

- `handle`: Device handle obtained using `spi_host_add_dev`
- `trans_desc`: Description of transaction to execute

`esp_err_t spi_device_polling_start` (`spi_device_handle_t handle`, `spi_transaction_t *trans_desc`, `TickType_t ticks_to_wait`)

Immediately start a polling transaction.

Note Normally a device cannot start (queue) polling and interrupt transactions simultaneously. Moreover, a device cannot start a new polling transaction if another polling transaction is not finished.

Return

- `ESP_ERR_INVALID_ARG` if parameter is invalid
- `ESP_ERR_TIMEOUT` if the device cannot get control of the bus before `ticks_to_wait` expired
- `ESP_ERR_NO_MEM` if allocating DMA-capable temporary buffer failed
- `ESP_ERR_INVALID_STATE` if previous transactions are not finished
- `ESP_OK` on success

Parameters

- `handle`: Device handle obtained using `spi_host_add_dev`
- `trans_desc`: Description of transaction to execute
- `ticks_to_wait`: Ticks to wait until there's room in the queue; currently only `portMAX_DELAY` is supported.

`esp_err_t spi_device_polling_end` (`spi_device_handle_t handle`, `TickType_t ticks_to_wait`)

Poll until the polling transaction ends.

This routine will not return until the transaction to the given device has successfully completed. The task is not blocked, but actively busy-spins for the transaction to be completed.

Return

- `ESP_ERR_INVALID_ARG` if parameter is invalid
- `ESP_ERR_TIMEOUT` if the transaction cannot finish before `ticks_to_wait` expired
- `ESP_OK` on success

Parameters

- `handle`: Device handle obtained using `spi_host_add_dev`
- `ticks_to_wait`: Ticks to wait until there's a returned item; use `portMAX_DELAY` to never time out.

`esp_err_t spi_device_polling_transmit` (`spi_device_handle_t handle`, `spi_transaction_t *trans_desc`)

Send a polling transaction, wait for it to complete, and return the result.

This function is the equivalent of calling `spi_device_polling_start()` followed by `spi_device_polling_end()`. Do not use this when there is still a transaction that hasn't been finalized.

Note This function is not thread safe when multiple tasks access the same SPI device. Normally a device cannot start (queue) polling and interrupt transactions simultaneously.

Return

- `ESP_ERR_INVALID_ARG` if parameter is invalid
- `ESP_OK` on success

Parameters

- `handle`: Device handle obtained using `spi_host_add_dev`
- `trans_desc`: Description of transaction to execute

`esp_err_t spi_device_acquire_bus` (`spi_device_handle_t device`, `TickType_t wait`)

Occupy the SPI bus for a device to do continuous transactions.

Transactions to all other devices will be put off until `spi_device_release_bus` is called.

Note The function will wait until all the existing transactions have been sent.

Return

- `ESP_ERR_INVALID_ARG`: `wait` is not set to `portMAX_DELAY`.
- `ESP_OK`: Success.

Parameters

- `device`: The device to occupy the bus.

- `wait`: Time to wait before the the bus is occupied by the device. Currently MUST set to `port-MAX_DELAY`.

void **spi_device_release_bus** (*spi_device_handle_t dev*)

Release the SPI bus occupied by the device. All other devices can start sending transactions.

Parameters

- `dev`: The device to release the bus.

int **spi_cal_clock** (int *fapb*, int *hz*, int *duty_cycle*, uint32_t **reg_o*)

Calculate the working frequency that is most close to desired frequency, and also the register value.

Parameters

- `fapb`: The frequency of apb clock, should be `APB_CLK_FREQ`.
- `hz`: Desired working frequency
- `duty_cycle`: Duty cycle of the spi clock
- `reg_o`: Output of value to be set in clock register, or NULL if not needed.

Return Actual working frequency that most fit.

int **spi_get_actual_clock** (int *fapb*, int *hz*, int *duty_cycle*)

Calculate the working frequency that is most close to desired frequency.

Return Actual working frequency that most fit.

Parameters

- `fapb`: The frequency of apb clock, should be `APB_CLK_FREQ`.
- `hz`: Desired working frequency
- `duty_cycle`: Duty cycle of the spi clock

void **spi_get_timing** (bool *gpio_is_used*, int *input_delay_ns*, int *eff_clk*, int **dummy_o*, int **cycles_remain_o*)

Calculate the timing settings of specified frequency and settings.

Note If `**dummy_o` is not zero, it means dummy bits should be applied in half duplex mode, and full duplex mode may not work.

Parameters

- `gpio_is_used`: True if using GPIO matrix, or False if iomux pins are used.
- `input_delay_ns`: Input delay from SCLK launch edge to MISO data valid.
- `eff_clk`: Effective clock frequency (in Hz) from `spi_cal_clock`.
- `dummy_o`: Address of dummy bits used output. Set to NULL if not needed.
- `cycles_remain_o`: Address of cycles remaining (after dummy bits are used) output.
 - -1 If too many cycles remaining, suggest to compensate half a clock.
 - 0 If no remaining cycles or dummy bits are not used.
 - positive value: cycles suggest to compensate.

int **spi_get_freq_limit** (bool *gpio_is_used*, int *input_delay_ns*)

Get the frequency limit of current configurations. SPI master working at this limit is OK, while above the limit, full duplex mode and DMA will not work, and dummy bits will be applied in the half duplex mode.

Return Frequency limit of current configurations.

Parameters

- `gpio_is_used`: True if using GPIO matrix, or False if native pins are used.
- `input_delay_ns`: Input delay from SCLK launch edge to MISO data valid.

Structures

struct spi_device_interface_config_t

This is a configuration for a SPI slave device that is connected to one of the SPI buses.

Public Members

uint8_t **command_bits**

Default amount of bits in command phase (0-16), used when `SPI_TRANS_VARIABLE_CMD` is not

used, otherwise ignored.

uint8_t address_bits

Default amount of bits in address phase (0-64), used when `SPI_TRANS_VARIABLE_ADDR` is not used, otherwise ignored.

uint8_t dummy_bits

Amount of dummy bits to insert between address and data phase.

uint8_t mode

SPI mode (0-3)

uint16_t duty_cycle_pos

Duty cycle of positive clock, in 1/256th increments (128 = 50%/50% duty). Setting this to 0 (=not setting it) is equivalent to setting this to 128.

uint16_t cs_ena_pretrans

Amount of SPI bit-cycles the cs should be activated before the transmission (0-16). This only works on half-duplex transactions.

uint8_t cs_ena_posttrans

Amount of SPI bit-cycles the cs should stay active after the transmission (0-16)

int clock_speed_hz

Clock speed, divisors of 80MHz, in Hz. See `SPI_MASTER_FREQ_*`.

int input_delay_ns

Maximum data valid time of slave. The time required between SCLK and MISO valid, including the possible clock delay from slave to master. The driver uses this value to give an extra delay before the MISO is ready on the line. Leave at 0 unless you know you need a delay. For better timing performance at high frequency (over 8MHz), it's suggest to have the right value.

int spics_io_num

CS GPIO pin for this device, or -1 if not used.

uint32_t flags

Bitwise OR of `SPI_DEVICE_*` flags.

int queue_size

Transaction queue size. This sets how many transactions can be 'in the air' (queued using `spi_device_queue_trans` but not yet finished using `spi_device_get_trans_result`) at the same time.

transaction_cb_t pre_cb

Callback to be called before a transmission is started.

This callback is called within interrupt context should be in IRAM for best performance, see "Transferring Speed" section in the SPI Master documentation for full details. If not, the callback may crash during flash operation when the driver is initialized with `ESP_INTR_FLAG_IRAM`.

transaction_cb_t post_cb

Callback to be called after a transmission has completed.

This callback is called within interrupt context should be in IRAM for best performance, see "Transferring Speed" section in the SPI Master documentation for full details. If not, the callback may crash during flash operation when the driver is initialized with `ESP_INTR_FLAG_IRAM`.

struct spi_transaction_t

This structure describes one SPI transaction. The descriptor should not be modified until the transaction finishes.

Public Members**uint32_t flags**

Bitwise OR of `SPI_TRANS_*` flags.

uint16_t cmd

Command data, of which the length is set in the `command_bits` of *spi_device_interface_config_t*.

NOTE: this field, used to be “command” in ESP-IDF 2.1 and before, is re-written to be used in a new way in ESP-IDF 3.0.

Example: write 0x0123 and `command_bits=12` to send command 0x12, 0x3_ (in previous version, you may have to write 0x3_12).

uint64_t addr

Address data, of which the length is set in the `address_bits` of *spi_device_interface_config_t*.

NOTE: this field, used to be “address” in ESP-IDF 2.1 and before, is re-written to be used in a new way in ESP-IDF3.0.

Example: write 0x123400 and `address_bits=24` to send address of 0x12, 0x34, 0x00 (in previous version, you may have to write 0x12340000).

size_t length

Total data length, in bits.

size_t rxlength

Total data length received, should be not greater than `length` in full-duplex mode (0 defaults this to the value of `length`).

void *user

User-defined variable. Can be used to store eg transaction ID.

const void *tx_buffer

Pointer to transmit buffer, or NULL for no MOSI phase.

uint8_t tx_data[4]

If `SPI_TRANS_USE_TXDATA` is set, data set here is sent directly from this variable.

void *rx_buffer

Pointer to receive buffer, or NULL for no MISO phase. Written by 4 bytes-unit if DMA is used.

uint8_t rx_data[4]

If `SPI_TRANS_USE_RXDATA` is set, data is received directly to this variable.

struct spi_transaction_ext_t

This struct is for SPI transactions which may change their address and command length. Please do set the flags in base to `SPI_TRANS_VARIABLE_CMD_ADR` to use the bit length here.

Public Members**struct spi_transaction_t base**

Transaction data, so that pointer to *spi_transaction_t* can be converted into *spi_transaction_ext_t*.

uint8_t command_bits

The command length in this transaction, in bits.

uint8_t address_bits

The address length in this transaction, in bits.

uint8_t dummy_bits

The dummy length in this transaction, in bits.

Macros**SPI_DEVICE_TXBIT_LSBFIRST**

Transmit command/address/data LSB first instead of the default MSB first.

SPI master clock is divided by 80MHz apb clock. Below defines are example frequencies, and are accurate. Be free to specify a random frequency, it will be rounded to closest frequency (to macros below if above 8MHz). 8MHz

SPI_DEVICE_RXBIT_LSBFIRST

Receive data LSB first instead of the default MSB first.

SPI_DEVICE_BIT_LSBFIRST

Transmit and receive LSB first.

SPI_DEVICE_3WIRE

Use MOSI (=spid) for both sending and receiving data.

SPI_DEVICE_POSITIVE_CS

Make CS positive during a transaction instead of negative.

SPI_DEVICE_HALFDUPLEX

Transmit data before receiving it, instead of simultaneously.

SPI_DEVICE_CLK_AS_CS

Output clock on CS line if CS is active.

SPI_DEVICE_NO_DUMMY

There are timing issue when reading at high frequency (the frequency is related to whether iomux pins are used, valid time after slave sees the clock).

- In half-duplex mode, the driver automatically inserts dummy bits before reading phase to fix the timing issue. Set this flag to disable this feature.
- In full-duplex mode, however, the hardware cannot use dummy bits, so there is no way to prevent data being read from getting corrupted. Set this flag to confirm that you're going to work with output only, or read without dummy bits at your own risk.

SPI_DEVICE_DDRCLK**SPI_TRANS_MODE_DIO**

Transmit/receive data in 2-bit mode.

SPI_TRANS_MODE_QIO

Transmit/receive data in 4-bit mode.

SPI_TRANS_USE_RXDATA

Receive into rx_data member of *spi_transaction_t* instead into memory at rx_buffer.

SPI_TRANS_USE_TXDATA

Transmit tx_data member of *spi_transaction_t* instead of data at tx_buffer. Do not set tx_buffer when using this.

SPI_TRANS_MODE_DIOQIO_ADDR

Also transmit address in mode selected by SPI_MODE_DIO/SPI_MODE_QIO.

SPI_TRANS_VARIABLE_CMD

Use the command_bits in *spi_transaction_ext_t* rather than default value in *spi_device_interface_config_t*.

SPI_TRANS_VARIABLE_ADDR

Use the address_bits in *spi_transaction_ext_t* rather than default value in *spi_device_interface_config_t*.

SPI_TRANS_VARIABLE_DUMMY

Use the dummy_bits in *spi_transaction_ext_t* rather than default value in *spi_device_interface_config_t*.

SPI_TRANS_SET_CD

Set the CD pin.

Type Definitions

```
typedef struct spi_transaction_t spi_transaction_t
typedef void (*transaction_cb_t) (spi_transaction_t *trans)
```

```
typedef struct spi_device_t *spi_device_handle_t  
    Handle for a device on a SPI bus.
```

2.3.17 SPI Slave Driver

SPI Slave driver is a program that controls ESP32' s SPI peripherals while they function as slaves.

Overview of ESP32' s SPI peripherals

ESP32 integrates two general purpose SPI controllers which can be used as slave nodes driven by an off-chip SPI master

- SPI2, sometimes referred to as HSPI
- SPI3, sometimes referred to as VSPI

SPI2 and SPI3 have independent signal buses with the same respective names.

Terminology

The terms used in relation to the SPI slave driver are given in the table below.

| Term | Definition |
|--------------------|--|
| Host | The SPI controller peripheral external to ESP32 that initiates SPI transmissions over the bus, and acts as an SPI Master. |
| Device | SPI slave device, in this case the SPI2 and SPI3 controllers. Each Device shares the MOSI, MISO and SCLK signals but is only active on the bus when the Host asserts the Device's individual CS line. |
| Bus | A signal bus, common to all Devices connected to one Host. In general, a bus includes the following lines: MISO, MOSI, SCLK, one or more CS lines, and, optionally, QUADWP and QUADHD. So Devices are connected to the same lines, with the exception that each Device has its own CS line. Several Devices can also share one CS line if connected in the daisy-chain manner. |
| • MISO | Master In, Slave Out, a.k.a. Q. Data transmission from a Device to Host. |
| • MOSI | Master Out, Slave in, a.k.a. D. Data transmission from a Host to Device. |
| • SCLK | Serial Clock. Oscillating signal generated by a Host that keeps the transmission of data bits in sync. |
| • CS | Chip Select. Allows a Host to select individual Device(s) connected to the bus in order to send or receive data. |
| • QUADWP | Write Protect signal. Only used for 4-bit (qio/qout) transactions. |
| • QUADHD | Hold signal. Only used for 4-bit (qio/qout) transactions. |
| • Assertion | The action of activating a line. The opposite action of returning the line back to inactive (back to idle) is called <i>de-assertion</i> . |
| Transaction | One instance of a Host asserting a CS line, transferring data to and from a Device, and de-asserting the CS line. Transactions are atomic, which means they can never be interrupted by another transaction. |
| Launch edge | Edge of the clock at which the source register <i>launches</i> the signal onto the line. |
| Latch edge | Edge of the clock at which the destination register <i>latches in</i> the signal. |

Driver Features

The SPI slave driver allows using the SPI2 and/or SPI3 peripherals as full-duplex Devices. The driver can send/receive transactions up to 64 bytes in length, or utilize DMA to send/receive longer transactions. However, there are some *known issues* related to DMA.

SPI Transactions

A full-duplex SPI transaction begins when the Host asserts the CS line and starts sending out clock pulses on the SCLK line. Every clock pulse, a data bit is shifted from the Host to the Device on the MOSI line and back on the MISO line at the same time. At the end of the transaction, the Host de-asserts the CS line.

The attributes of a transaction are determined by the configuration structure for an SPI host acting as a slave device `spi_slave_interface_config_t`, and transaction configuration structure `spi_slave_transaction_t`.

As not every transaction requires both writing and reading data, you have a choice to configure the `spi_transaction_t` structure for TX only, RX only, or TX and RX transactions. If `spi_slave_transaction_t::rx_buffer` is set to NULL, the read phase will be skipped. If `spi_slave_transaction_t::tx_buffer` is set to NULL, the write phase will be skipped.

Note: A Host should not start a transaction before its Device is ready for receiving data. It is recommended to use another GPIO pin for a handshake signal to sync the Devices. For more details, see [Transaction Interval](#).

Driver Usage

- Initialize an SPI peripheral as a Device by calling the function `cpp:func:spi_slave_initialize`. Make sure to set the correct I/O pins in the struct `bus_config`. Set the unused signals to -1. If transactions will be longer than 32 bytes, allow a DMA channel 1 or 2 by setting the parameter `dma_chan` to 1 or 2 respectively. Otherwise, set `dma_chan` to 0.
- Before initiating transactions, fill one or more `spi_slave_transaction_t` structs with the transaction parameters required. Either queue all transactions by calling the function `spi_slave_queue_trans()` and, at a later time, query the result by using the function `spi_slave_get_trans_result()`, or handle all requests individually by feeding them into `spi_slave_transmit()`. The latter two functions will be blocked until the Host has initiated and finished a transaction, causing the queued data to be sent and received.
- (Optional) To unload the SPI slave driver, call `spi_slave_free()`.

Transaction Data and Master/Slave Length Mismatches

Normally, the data that needs to be transferred to or from a Device is read or written to a chunk of memory indicated by the `rx_buffer` and `tx_buffer` members of the `spi_transaction_t` structure. The SPI driver can be configured to use DMA for transfers, in which case these buffers must be allocated in DMA-capable memory using `pvPortMallocCaps(size, MALLOC_CAP_DMA)`.

The amount of data that the driver can read or write to the buffers is limited by the member `spi_transaction_t::length`. However, this member does not define the actual length of an SPI transaction. A transaction's length is determined by a Host which drives the clock and CS lines. The actual length of the transmission can be read only after a transaction is finished from the member `spi_slave_transaction_t::trans_len`.

If the length of the transmission is greater than the buffer length, only the initial number of bits specified in the `length` member will be sent and received. In this case, `trans_len` is set to `length` instead of the actual transaction length. To meet the actual transaction length requirements, set `length` to a value greater than the maximum `trans_len` expected. If the transmission length is shorter than the buffer length, only the data equal to the length of the buffer will be transmitted.

Warning: The ESP32 DMA hardware has a limit to the number of bytes sent by a Host and received by a Device. The transaction length must be longer than 8 bytes and a multiple of 4 bytes; otherwise, the SPI hardware might fail to receive the last 1 to 7 bytes.

GPIO Matrix and IO_MUX

Most of ESP32's peripheral signals have direct connection to their dedicated IO_MUX pins. However, the signals can also be routed to any other available pins using the less direct GPIO matrix.

If at least one signal is routed through the GPIO matrix, then all signals will be routed through it. The GPIO matrix samples all signals at 80 MHz and transmits them between the GPIO and the peripheral.

If the driver is configured so that all SPI signals are either routed to their dedicated IO_MUX pins or are not connected at all, the GPIO matrix will be bypassed.

The GPIO matrix introduces flexibility of routing but also increases the input delay of the MISO signal, which makes MISO setup time violations more likely. If SPI needs to operate at high speeds, use dedicated IO_MUX pins.

Note: For more details about the influence of the MISO input delay on the maximum clock frequency, see [Timing Considerations](#).

The IO_MUX pins for SPI buses are given below.

| Pin Name | SPI2 | SPI3 |
|----------|-------------|------|
| | GPIO Number | |
| CS0* | 15 | 5 |
| SCLK | 14 | 18 |
| MISO | 12 | 19 |
| MOSI | 13 | 23 |
| QUADWP | 2 | 22 |
| QUADHD | 4 | 21 |

- Only the first Device attached to the bus can use the CS0 pin.

Speed and Timing Considerations

Transaction Interval The ESP32 SPI slave peripherals are designed as general purpose Devices controlled by a CPU. As opposed to dedicated slaves, CPU-based SPI Devices have a limited number of pre-defined registers. All transactions must be handled by the CPU, which means that the transfers and responses are not real-time, and there might be noticeable latency.

As a solution, a Device's response rate can be doubled by using the functions `spi_slave_queue_trans()` and then `spi_slave_get_trans_result()` instead of using `spi_slave_transmit()`.

You can also configure a GPIO pin through which the Device will signal to the Host when it is ready for a new transaction. A code example of this can be found in [peripherals/spi_slave](#).

SCLK Frequency Requirements The SPI slaves are designed to operate at up to 10 MHz. The data cannot be recognized or received correctly if the clock is too fast or does not have a 50% duty cycle.

On top of that, there are additional requirements for the data to meet the timing constraints:

- **Read (MOSI):** The Device can read data correctly only if the data is already set at the launch edge. Although it is usually the case for most masters.
- **Write (MISO):** The output delay of the MISO signal needs to be shorter than half of a clock cycle period so that the MISO line is stable before the next latch edge. Given that the clock is balanced, the output delay and frequency limitations in different cases are given below.

| | Output delay of MISO (ns) | Freq. limit (MHz) |
|-------------|---------------------------|-------------------|
| IO_MUX | 43.75 | <11.4 |
| GPIO matrix | 68.75 | <7.2 |

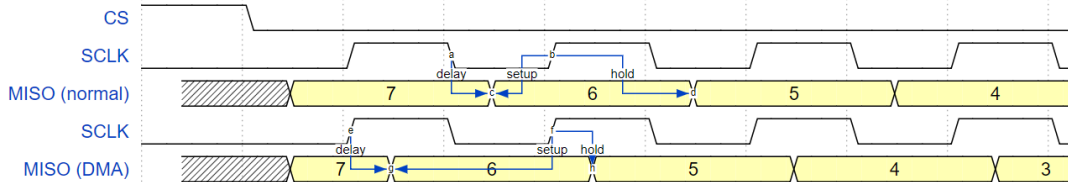
Note:

1. If the frequency is equal to the limitation, it can lead to random errors.
2. The clock uncertainty between Host and Device (12.5ns) is included.
3. The output delay is measured under ideal circumstances (no load). If the MISO pin is heavily loaded, the output delay will be longer, and the maximum allowed frequency will be lower.

Exception: The frequency is allowed to be higher if the master has more tolerance for the MISO setup time, e.g., latch data at the next edge than expected, or configurable latching time.

Restrictions and Known Issues

1. If DMA is enabled, the rx buffer should be word-aligned (starting from a 32-bit boundary and having a length of multiples of 4 bytes). Otherwise, DMA may write incorrectly or not in a boundary aligned manner. The driver reports an error if this condition is not satisfied.
Also, a Host should write lengths that are multiples of 4 bytes. The data with inappropriate lengths will be discarded.
2. Furthermore, DMA requires SPI modes 1 and 3. For SPI modes 0 and 2, the MISO signal has to be launched half a clock cycle earlier to meet the timing. The new timing is as follows:



If DMA is enabled, a Device's launch edge is half of an SPI clock cycle ahead of the normal time, shifting to the Master's actual latch edge. In this case, if the GPIO matrix is bypassed, the hold time for data sampling is 68.75 ns and no longer a half of an SPI clock cycle. If the GPIO matrix is used, the hold time will increase to 93.75 ns. The Host should sample the data immediately at the latch edge or communicate in SPI modes 1 or 3. If your Host cannot meet these timing requirements, initialize your Device without DMA.

Application Example

The code example for Device/Host communication can be found in the [peripherals/spi_slave](#) directory of ESP-IDF examples.

API Reference

Header File

- [driver/include/driver/spi_slave.h](#)

Functions

```
esp_err_t spi_slave_initialize(spi_host_device_t host, const spi_bus_config_t *bus_config,
                             const spi_slave_interface_config_t *slave_config, int dma_chan)
```

Initialize a SPI bus as a slave interface.

Warning For now, only supports HSPI and VSPI.

Warning If a DMA channel is selected, any transmit and receive buffer used should be allocated in DMA-capable memory.

Warning The ISR of SPI is always executed on the core which calls this function. Never starve the ISR on this core or the SPI transactions will not be handled.

Return

- `ESP_ERR_INVALID_ARG` if configuration is invalid
- `ESP_ERR_INVALID_STATE` if host already is in use
- `ESP_ERR_NO_MEM` if out of memory
- `ESP_OK` on success

Parameters

- `host`: SPI peripheral to use as a SPI slave interface
- `bus_config`: Pointer to a [spi_bus_config_t](#) struct specifying how the host should be initialized
- `slave_config`: Pointer to a [spi_slave_interface_config_t](#) struct specifying the details for the slave interface
- `dma_chan`: Either 1 or 2. A SPI bus used by this driver must have a DMA channel associated with it. The SPI hardware has two DMA channels to share. This parameter indicates which one to use.

```
esp_err_t spi_slave_free(spi_host_device_t host)
```

Free a SPI bus claimed as a SPI slave interface.

Return

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_INVALID_STATE if not all devices on the bus are freed
- ESP_OK on success

Parameters

- *host*: SPI peripheral to free

esp_err_t **spi_slave_queue_trans** (*spi_host_device_t* *host*, **const** *spi_slave_transaction_t* **trans_desc*, TickType_t *ticks_to_wait*)

Queue a SPI transaction for execution.

Queues a SPI transaction to be executed by this slave device. (The transaction queue size was specified when the slave device was initialised via `spi_slave_initialize`.) This function may block if the queue is full (depending on the `ticks_to_wait` parameter). No SPI operation is directly initiated by this function, the next queued transaction will happen when the master initiates a SPI transaction by pulling down CS and sending out clock signals.

This function hands over ownership of the buffers in `trans_desc` to the SPI slave driver; the application is not to access this memory until `spi_slave_queue_trans` is called to hand ownership back to the application.

Return

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_OK on success

Parameters

- *host*: SPI peripheral that is acting as a slave
- *trans_desc*: Description of transaction to execute. Not const because we may want to write status back into the transaction description.
- *ticks_to_wait*: Ticks to wait until there's room in the queue; use `portMAX_DELAY` to never time out.

esp_err_t **spi_slave_get_trans_result** (*spi_host_device_t* *host*, *spi_slave_transaction_t* ***trans_desc*, TickType_t *ticks_to_wait*)

Get the result of a SPI transaction queued earlier.

This routine will wait until a transaction to the given device (queued earlier with `spi_slave_queue_trans`) has successfully completed. It will then return the description of the completed transaction so software can inspect the result and e.g. free the memory or re-use the buffers.

It is mandatory to eventually use this function for any transaction queued by `spi_slave_queue_trans`.

Return

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_OK on success

Parameters

- *host*: SPI peripheral to that is acting as a slave
- [*out*] *trans_desc*: Pointer to variable able to contain a pointer to the description of the transaction that is executed
- *ticks_to_wait*: Ticks to wait until there's a returned item; use `portMAX_DELAY` to never time out.

esp_err_t **spi_slave_transmit** (*spi_host_device_t* *host*, *spi_slave_transaction_t* **trans_desc*, TickType_t *ticks_to_wait*)

Do a SPI transaction.

Essentially does the same as `spi_slave_queue_trans` followed by `spi_slave_get_trans_result`. Do not use this when there is still a transaction queued that hasn't been finalized using `spi_slave_get_trans_result`.

Return

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_OK on success

Parameters

- *host*: SPI peripheral to that is acting as a slave
- *trans_desc*: Pointer to variable able to contain a pointer to the description of the transaction that is executed. Not const because we may want to write status back into the transaction description.

- `ticks_to_wait`: Ticks to wait until there's a returned item; use `portMAX_DELAY` to never time out.

Structures

struct spi_slave_interface_config_t

This is a configuration for a SPI host acting as a slave device.

Public Members

int **spics_io_num**

CS GPIO pin for this device.

uint32_t **flags**

Bitwise OR of `SPI_SLAVE_*` flags.

int **queue_size**

Transaction queue size. This sets how many transactions can be 'in the air' (queued using `spi_slave_queue_trans` but not yet finished using `spi_slave_get_trans_result`) at the same time.

uint8_t **mode**

SPI mode (0-3)

slave_transaction_cb_t **post_setup_cb**

Callback called after the SPI registers are loaded with new data.

This callback is called within interrupt context should be in IRAM for best performance, see "Transferring Speed" section in the SPI Master documentation for full details. If not, the callback may crash during flash operation when the driver is initialized with `ESP_INTR_FLAG_IRAM`.

slave_transaction_cb_t **post_trans_cb**

Callback called after a transaction is done.

This callback is called within interrupt context should be in IRAM for best performance, see "Transferring Speed" section in the SPI Master documentation for full details. If not, the callback may crash during flash operation when the driver is initialized with `ESP_INTR_FLAG_IRAM`.

struct spi_slave_transaction_t

This structure describes one SPI transaction

Public Members

size_t **length**

Total data length, in bits.

size_t **trans_len**

Transaction data length, in bits.

const void ***tx_buffer**

Pointer to transmit buffer, or NULL for no MOSI phase.

void ***rx_buffer**

Pointer to receive buffer, or NULL for no MISO phase. When the DMA is enabled, must start at WORD boundary (`rx_buffer%4==0`), and has length of a multiple of 4 bytes.

void ***user**

User-defined variable. Can be used to store eg transaction ID.

Macros

SPI_SLAVE_TXBIT_LSBFIRST

Transmit command/address/data LSB first instead of the default MSB first.

SPI_SLAVE_RXBIT_LSBFIRST

Receive data LSB first instead of the default MSB first.

SPI_SLAVE_BIT_LSBFIRST

Transmit and receive LSB first.

Type Definitions

```
typedef struct spi_slave_transaction_t spi_slave_transaction_t
typedef void (*slave_transaction_cb_t) (spi_slave_transaction_t *trans)
```

2.3.18 ESP32-S2 Temperature Sensor

Overview

The ESP32-S2 has a built-in temperature sensor. The temperature sensor module contains an 8-bit Sigma-Delta ADC and a temperature offset DAC. The conversion relationship is the first columns of the table below. Among them, offset = 0 is the main measurement option, and other values are extended measurement options.

| offset | measure range(Celsius) | measure error(Celsius) |
|--------|------------------------|------------------------|
| -2 | 50 ~ 125 | < 3 |
| -1 | 20 ~ 100 | < 2 |
| 0 | -10 ~ 80 | < 1 |
| 1 | -30 ~ 50 | < 2 |
| 2 | -40 ~ 20 | < 3 |

Application Example

Temperature sensor reading example: [peripherals/temp_sensor_esp32s2](#).

API Reference - Normal Temp Sensor**Header File**

- [driver/esp32s2beta/include/temp_sensor.h](#)

Functions

esp_err_t **temp_sensor_set_config** (*temp_sensor_config_t* *tsens*)

Set parameter of temperature sensor.

Return

- ESP_OK Success

Parameters

- *tsens*:

esp_err_t **temp_sensor_get_config** (*temp_sensor_config_t* **tsens*)

Get parameter of temperature sensor.

Return

- ESP_OK Success

Parameters

- *tsens*:

esp_err_t **temp_sensor_start** (void)

Start temperature sensor measure.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG

esp_err_t **temp_sensor_stop** (void)

Stop temperature sensor measure.

Return

- ESP_OK Success

esp_err_t **temp_sensor_read_raw** (uint32_t **tsens_out*)

Read temperature sensor raw data.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG *tsens_out* is NULL
- ESP_ERR_INVALID_STATE temperature sensor dont start

Parameters

- *tsens_out*: Pointer to raw data, Range: 0 ~ 255

esp_err_t **temp_sensor_read_celsius** (float **celsius*)

Read temperature sensor data that is converted to degrees Celsius.

Note Should not be called from interrupt.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG ARG is NULL.
- ESP_ERR_INVALID_STATE The ambient temperature is out of range.

Parameters

- *celsius*: The measure output value.

Structures

struct temp_sensor_config_t

Configuration for temperature sensor reading.

Public Members

temp_sensor_dac_offset_t **dac_offset**

The temperature measurement range is configured with a built-in temperature offset DAC.

uint8_t **clk_div**

Default: 6

Macros

TSENS_CONFIG_DEFAULT ()

Enumerations

enum temp_sensor_dac_offset_t

Values:

TSENS_DAC_L0 = 0

offset = -2, measure range: 50°C ~ 125°C, error < 3°C.

TSENS_DAC_L1

offset = -1, measure range: 20°C ~ 100°C, error < 2°C.

TSENS_DAC_L2

offset = 0, measure range:-10°C ~ 80°C, error < 1°C.

TSENS_DAC_L3

offset = 1, measure range:-30°C ~ 50°C, error < 2°C.

TSENS_DAC_L4

offset = 2, measure range:-40°C ~ 20°C, error < 3°C.

TSENS_DAC_MAX

TSENS_DAC_DEFAULT = *TSENS_DAC_L2*

2.3.19 Timer

Introduction

The ESP32 chip contains two hardware timer groups. Each group has two general-purpose hardware timers. They are all 64-bit generic timers based on 16-bit prescalers and 64-bit up / down counters which are capable of being auto-reloaded.

Functional Overview

The following sections of this document cover the typical steps to configure and operate a timer:

- *Timer Initialization* - covers which parameters should be set up to get the timer working, and also what specific functionality is provided depending on the timer configuration.
- *Timer Control* - describes how to read a timer's value, pause or start a timer, and change how it operates.
- *Alarms* - shows how to set and use alarms.
- *Interrupts* - explains how to enable and use interrupts.

Timer Initialization The two ESP32 timer groups, with two timers in each, provide the total of four individual timers for use. An ESP32 timer group should be identified using `timer_group_t`. An individual timer in a group should be identified with `timer_idx_t`.

First of all, the timer should be initialized by calling the function `timer_init()` and passing a structure `timer_config_t` to it to define how the timer should operate. In particular, the following timer parameters can be set:

- **Divider:** Sets how quickly the timer's counter is "ticking". The setting `divider` is used as a divisor of the incoming 80 MHz `APB_CLK` clock.
- **Mode:** Sets if the counter should be incrementing or decrementing. It can be defined using `counter_dir` by selecting one of the values from `timer_count_dir_t`.
- **Counter Enable:** If the counter is enabled, it will start incrementing / decrementing immediately after calling `timer_init()`. You can change the behavior with `counter_en` by selecting one of the values from `timer_start_t`.
- **Alarm Enable:** Can be set using `alarm_en`.
- **Auto Reload:** Sets if the counter should `auto_reload` the initial counter value on the timer's alarm or continue incrementing or decrementing.
- **Interrupt Type:** Select which interrupt type should be triggered on the timer's alarm. Set the value defined in `timer_intr_mode_t`.

To get the current values of the timer's settings, use the function `timer_get_config()`.

Timer Control Once the timer is enabled, its counter starts running. To enable the timer, call the function `timer_init()` with `counter_en` set to `true`, or call `timer_start()`. You can specify the timer's initial counter value by calling `timer_set_counter_value()`. To check the timer's current value, call `timer_get_counter_value()` or `timer_get_counter_time_sec()`.

To pause the timer at any time, call `timer_pause()`. To resume it, call `timer_start()`.

To reconfigure the timer, you can call `timer_init()`. This function is described in Section *Timer Initialization*.

You can also reconfigure the timer by using dedicated functions to change individual settings:

| Setting | Dedicated Function | Description |
|-------------|---------------------------------------|---|
| Divider | <code>timer_set_divider()</code> | Change the rate of ticking. To avoid unpredictable results, the timer should be paused when changing the divider. If the timer is running, <code>timer_set_divider()</code> pauses it, change the setting, and start the timer again. |
| Mode | <code>timer_set_counter_mode()</code> | Set if the counter should be incrementing or decrementing |
| Auto Reload | <code>timer_set_auto_reload()</code> | Set if the initial counter value should be reloaded on the timer's alarm |

Alarms To set an alarm, call the function `timer_set_alarm_value()` and then enable the alarm using `timer_set_alarm()`. The alarm can also be enabled during the timer initialization stage, when `timer_init()` is called.

After the alarm is enabled, and the timer reaches the alarm value, the following two actions can occur depending on the configuration:

- An interrupt will be triggered if previously configured. See Section [Interrupts](#) on how to configure interrupts.
- When `auto_reload` is enabled, the timer's counter will automatically be reloaded to start counting again from a previously configured value. This value should be set in advance with `timer_set_counter_value()`.

Note:

- If an alarm value is set and the timer has already reached this value, the alarm is triggered immediately.
 - Once triggered, the alarm is disabled automatically and needs to be re-enabled to trigger again.
-

To check the specified alarm value, call `timer_get_alarm_value()`.

Interrupts Registration of the interrupt handler for a specific timer or a timer group can be done by calling `timer_isr_register()`.

To enable interrupts for a timer group, call `timer_group_intr_enable()`, for a specific timer call `timer_enable_intr()`. To disable interrupts for a timer group, call `timer_group_intr_disable()`, for a specified timer, call `timer_disable_intr()`.

When handling an interrupt within an interrupt service routine (ISR), the interrupt status bit needs to be explicitly cleared. To do that, set the `TIMERGN.int_clr_timers.tM` structure, defined in `soc/esp32/include/soc/timer_group_struct.h`. In this structure, N is the timer group number [0, 1], M is the timer number [0, 1]. For example, to clear an interrupt status bit for the timer 1 in the timer group 0, call the following:

```
TIMERG0.int_clr_timers.t1 = 1
```

For more information on how to use interrupts, please see the application example below.

Application Example

The 64-bit hardware timer example: [peripherals/timer_group](#).

API Reference

Header File

- `driver/include/driver/timer.h`

Functions

esp_err_t **timer_get_counter_value** (*timer_group_t* group_num, *timer_idx_t* timer_num, uint64_t *timer_val)

Read the counter value of hardware timer.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- group_num: Timer group, 0 for TIMERG0 or 1 for TIMERG1
- timer_num: Timer index, 0 for hw_timer[0] & 1 for hw_timer[1]
- timer_val: Pointer to accept timer counter value.

esp_err_t **timer_get_counter_time_sec** (*timer_group_t* group_num, *timer_idx_t* timer_num, double *time)

Read the counter value of hardware timer, in unit of a given scale.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- group_num: Timer group, 0 for TIMERG0 or 1 for TIMERG1
- timer_num: Timer index, 0 for hw_timer[0] & 1 for hw_timer[1]
- time: Pointer, type of double*, to accept timer counter value, in seconds.

esp_err_t **timer_set_counter_value** (*timer_group_t* group_num, *timer_idx_t* timer_num, uint64_t load_val)

Set counter value to hardware timer.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- group_num: Timer group, 0 for TIMERG0 or 1 for TIMERG1
- timer_num: Timer index, 0 for hw_timer[0] & 1 for hw_timer[1]
- load_val: Counter value to write to the hardware timer.

esp_err_t **timer_start** (*timer_group_t* group_num, *timer_idx_t* timer_num)

Start the counter of hardware timer.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- group_num: Timer group number, 0 for TIMERG0 or 1 for TIMERG1
- timer_num: Timer index, 0 for hw_timer[0] & 1 for hw_timer[1]

esp_err_t **timer_pause** (*timer_group_t* group_num, *timer_idx_t* timer_num)

Pause the counter of hardware timer.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- group_num: Timer group number, 0 for TIMERG0 or 1 for TIMERG1
- timer_num: Timer index, 0 for hw_timer[0] & 1 for hw_timer[1]

esp_err_t **timer_set_counter_mode** (*timer_group_t* group_num, *timer_idx_t* timer_num, *timer_count_dir_t* counter_dir)

Set counting mode for hardware timer.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `group_num`: Timer group number, 0 for `TIMERG0` or 1 for `TIMERG1`
- `timer_num`: Timer index, 0 for `hw_timer[0]` & 1 for `hw_timer[1]`
- `counter_dir`: Counting direction of timer, count-up or count-down

`esp_err_t timer_set_auto_reload(timer_group_t group_num, timer_idx_t timer_num, timer_autoreload_t reload)`

Enable or disable counter reload function when alarm event occurs.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `group_num`: Timer group number, 0 for `TIMERG0` or 1 for `TIMERG1`
- `timer_num`: Timer index, 0 for `hw_timer[0]` & 1 for `hw_timer[1]`
- `reload`: Counter reload mode.

`esp_err_t timer_set_divider(timer_group_t group_num, timer_idx_t timer_num, uint32_t divider)`

Set hardware timer source clock divider. Timer groups clock are divider from APB clock.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `group_num`: Timer group number, 0 for `TIMERG0` or 1 for `TIMERG1`
- `timer_num`: Timer index, 0 for `hw_timer[0]` & 1 for `hw_timer[1]`
- `divider`: Timer clock divider value. The divider's range is from 2 to 65536.

`esp_err_t timer_set_alarm_value(timer_group_t group_num, timer_idx_t timer_num, uint64_t alarm_value)`

Set timer alarm value.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `group_num`: Timer group, 0 for `TIMERG0` or 1 for `TIMERG1`
- `timer_num`: Timer index, 0 for `hw_timer[0]` & 1 for `hw_timer[1]`
- `alarm_value`: A 64-bit value to set the alarm value.

`esp_err_t timer_get_alarm_value(timer_group_t group_num, timer_idx_t timer_num, uint64_t *alarm_value)`

Get timer alarm value.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `group_num`: Timer group, 0 for `TIMERG0` or 1 for `TIMERG1`
- `timer_num`: Timer index, 0 for `hw_timer[0]` & 1 for `hw_timer[1]`
- `alarm_value`: Pointer of A 64-bit value to accept the alarm value.

`esp_err_t timer_set_alarm(timer_group_t group_num, timer_idx_t timer_num, timer_alarm_t alarm_en)`

Enable or disable generation of timer alarm events.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `group_num`: Timer group, 0 for `TIMERG0` or 1 for `TIMERG1`
- `timer_num`: Timer index, 0 for `hw_timer[0]` & 1 for `hw_timer[1]`
- `alarm_en`: To enable or disable timer alarm function.

esp_err_t **timer_isr_callback_add** (*timer_group_t* group_num, *timer_idx_t* timer_num, *timer_isr_t* isr_handler, void *arg, int intr_alloc_flags)

Add ISR handle callback for the corresponding timer.

The callback should return a bool value to determine whether need to do YIELD at the end of the ISR.

Note This ISR handler will be called from an ISR. This ISR handler do not need to handle interrupt status, and should be kept short. If you want to realize some specific applications or write the whole ISR, you can call `timer_isr_register(...)` to register ISR.

Parameters

- group_num: Timer group number
- timer_num: Timer index of timer group
- isr_handler: Interrupt handler function, it is a callback function.
- arg: Parameter for handler function
- intr_alloc_flags: Flags used to allocate the interrupt. One or multiple (ORred) ESP_INTR_FLAG_* values. See `esp_intr_alloc.h` for more info.

If the `intr_alloc_flags` value `ESP_INTR_FLAG_IRAM` is set, the handler function must be declared with `IRAM_ATTR` attribute and can only call functions in IRAM or ROM. It cannot call other timer APIs.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **timer_isr_callback_remove** (*timer_group_t* group_num, *timer_idx_t* timer_num)

Remove ISR handle callback for the corresponding timer.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- group_num: Timer group number
- timer_num: Timer index of timer group

esp_err_t **timer_isr_register** (*timer_group_t* group_num, *timer_idx_t* timer_num, void (*fn)() void *, void *arg, int intr_alloc_flags, *timer_isr_handle_t* *handle) Register Timer interrupt handler, the handler is an ISR. The handler will be attached to the same CPU core that this function is running on.

If the `intr_alloc_flags` value `ESP_INTR_FLAG_IRAM` is set, the handler function must be declared with `IRAM_ATTR` attribute and can only call functions in IRAM or ROM. It cannot call other timer APIs. Use direct register access to configure timers from inside the ISR in this case.

Note If use this function to register ISR, you need to write the whole ISR. In the interrupt handler, you need to call `timer_spinlock_take(...)` before your handling, and call `timer_spinlock_give(...)` after your handling.

Parameters

- group_num: Timer group number
- timer_num: Timer index of timer group
- fn: Interrupt handler function.
- arg: Parameter for handler function
- intr_alloc_flags: Flags used to allocate the interrupt. One or multiple (ORred) ESP_INTR_FLAG_* values. See `esp_intr_alloc.h` for more info.
- handle: Pointer to return handle. If non-NULL, a handle for the interrupt will be returned here.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **timer_init** (*timer_group_t* group_num, *timer_idx_t* timer_num, const *timer_config_t* *config)

Initializes and configure the timer.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `group_num`: Timer group number, 0 for `TIMERG0` or 1 for `TIMERG1`
- `timer_num`: Timer index, 0 for `hw_timer[0]` & 1 for `hw_timer[1]`
- `config`: Pointer to timer initialization parameters.

esp_err_t **timer_deinit** (*timer_group_t* group_num, *timer_idx_t* timer_num)

Deinitializes the timer.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `group_num`: Timer group number, 0 for `TIMERG0` or 1 for `TIMERG1`
- `timer_num`: Timer index, 0 for `hw_timer[0]` & 1 for `hw_timer[1]`

esp_err_t **timer_get_config** (*timer_group_t* group_num, *timer_idx_t* timer_num, *timer_config_t* *config)

Get timer configure value.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `group_num`: Timer group number, 0 for `TIMERG0` or 1 for `TIMERG1`
- `timer_num`: Timer index, 0 for `hw_timer[0]` & 1 for `hw_timer[1]`
- `config`: Pointer of struct to accept timer parameters.

esp_err_t **timer_group_intr_enable** (*timer_group_t* group_num, *timer_intr_t* intr_mask)

Enable timer group interrupt, by enable mask.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `group_num`: Timer group number, 0 for `TIMERG0` or 1 for `TIMERG1`
- `intr_mask`: Timer interrupt enable mask.
 - `TIMER_INTR_T0`: t0 interrupt
 - `TIMER_INTR_T1`: t1 interrupt
 - `TIMER_INTR_WDT`: watchdog interrupt

esp_err_t **timer_group_intr_disable** (*timer_group_t* group_num, *timer_intr_t* intr_mask)

Disable timer group interrupt, by disable mask.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `group_num`: Timer group number, 0 for `TIMERG0` or 1 for `TIMERG1`
- `intr_mask`: Timer interrupt disable mask.
 - `TIMER_INTR_T0`: t0 interrupt
 - `TIMER_INTR_T1`: t1 interrupt
 - `TIMER_INTR_WDT`: watchdog interrupt

esp_err_t **timer_enable_intr** (*timer_group_t* group_num, *timer_idx_t* timer_num)

Enable timer interrupt.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `group_num`: Timer group number, 0 for `TIMERG0` or 1 for `TIMERG1`
- `timer_num`: Timer index.

esp_err_t **timer_disable_intr** (*timer_group_t* group_num, *timer_idx_t* timer_num)

Disable timer interrupt.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `group_num`: Timer group number, 0 for TIMERG0 or 1 for TIMERG1
- `timer_num`: Timer index.

void `timer_group_intr_clr_in_isr` (*timer_group_t* `group_num`, *timer_idx_t* `timer_num`)

Clear timer interrupt status, just used in ISR.

Return

- None

Parameters

- `group_num`: Timer group number, 0 for TIMERG0 or 1 for TIMERG1
- `timer_num`: Timer index.

void `timer_group_clr_intr_status_in_isr` (*timer_group_t* `group_num`, *timer_idx_t* `timer_num`)

Clear timer interrupt status, just used in ISR.

Return

- None

Parameters

- `group_num`: Timer group number, 0 for TIMERG0 or 1 for TIMERG1
- `timer_num`: Timer index.

void `timer_group_enable_alarm_in_isr` (*timer_group_t* `group_num`, *timer_idx_t* `timer_num`)

Enable alarm interrupt, just used in ISR.

Return

- None

Parameters

- `group_num`: Timer group number, 0 for TIMERG0 or 1 for TIMERG1
- `timer_num`: Timer index.

uint64_t `timer_group_get_counter_value_in_isr` (*timer_group_t* `group_num`, *timer_idx_t* `timer_num`)

Get the current counter value, just used in ISR.

Return

- Counter value

Parameters

- `group_num`: Timer group number, 0 for TIMERG0 or 1 for TIMERG1
- `timer_num`: Timer index.

void `timer_group_set_alarm_value_in_isr` (*timer_group_t* `group_num`, *timer_idx_t* `timer_num`,
uint64_t `alarm_val`)

Set the alarm threshold for the timer, just used in ISR.

Return

- None

Parameters

- `group_num`: Timer group number, 0 for TIMERG0 or 1 for TIMERG1
- `timer_num`: Timer index.
- `alarm_val`: Alarm threshold.

void `timer_group_set_counter_enable_in_isr` (*timer_group_t* `group_num`, *timer_idx_t* `timer_num`,
timer_start_t `counter_en`)

Enable/disable a counter, just used in ISR.

Return

- None

Parameters

- `group_num`: Timer group number, 0 for TIMERG0 or 1 for TIMERG1
- `timer_num`: Timer index.

- `counter_en`: Enable/disable.

`timer_intr_t timer_group_intr_get_in_isr (timer_group_t group_num)`

Get the masked interrupt status, just used in ISR.

Return

- Interrupt status

Parameters

- `group_num`: Timer group number, 0 for `TIMERG0` or 1 for `TIMERG1`

`uint32_t timer_group_get_intr_status_in_isr (timer_group_t group_num)`

Get interrupt status, just used in ISR.

Return

- Interrupt status

Parameters

- `group_num`: Timer group number, 0 for `TIMERG0` or 1 for `TIMERG1`

`void timer_group_clr_intr_sta_in_isr (timer_group_t group_num, timer_intr_t intr_mask)`

Clear the masked interrupt status, just used in ISR.

Return

- None

Parameters

- `group_num`: Timer group number, 0 for `TIMERG0` or 1 for `TIMERG1`
- `intr_mask`: Masked interrupt.

`bool timer_group_get_auto_reload_in_isr (timer_group_t group_num, timer_idx_t timer_num)`

Get auto reload enable status, just used in ISR.

Return

- True Auto reload enabled
- False Auto reload disabled

Parameters

- `group_num`: Timer group number, 0 for `TIMERG0` or 1 for `TIMERG1`
- `timer_num`: Timer index

`esp_err_t timer_spinlock_take (timer_group_t group_num)`

Take timer spinlock to enter critical protect.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `group_num`: Timer group number, 0 for `TIMERG0` or 1 for `TIMERG1`

`esp_err_t timer_spinlock_give (timer_group_t group_num)`

Give timer spinlock to exit critical protect.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

Parameters

- `group_num`: Timer group number, 0 for `TIMERG0` or 1 for `TIMERG1`

Macros

`TIMER_BASE_CLK`

Frequency of the clock on the input of the timer groups

Type Definitions

`typedef bool (*timer_isr_t) (void *)`

Interrupt handle callback function. User need to retrun a bool value in callback.

Return

- True Do task yield at the end of ISR
- False Not do task yield at the end of ISR

Note If you called FreeRTOS functions in callback, you need to return true or false based on the return value of argument `pxHigherPriorityTaskWoken`. For example, `xQueueSendFromISR` is called in callback, if the return value `pxHigherPriorityTaskWoken` of any FreeRTOS calls is `pdTRUE`, return true; otherwise return false.

typedef *intr_handle_t* **timer_isr_handle_t**

Interrupt handle, used in order to free the isr after use. Aliases to an int handle for now.

Header File

- [soc/include/hal/timer_types.h](#)

Structures

struct **timer_config_t**

Data structure with timer's configuration settings.

Public Members

timer_alarm_t **alarm_en**

Timer alarm enable

timer_start_t **counter_en**

Counter enable

timer_intr_mode_t **intr_type**

Interrupt mode

timer_count_dir_t **counter_dir**

Counter direction

timer_autoreload_t **auto_reload**

Timer auto-reload

uint32_t **divider**

Counter clock divider. The divider's range is from 2 to 65536.

Enumerations

enum **timer_group_t**

Selects a Timer-Group out of 2 available groups.

Values:

TIMER_GROUP_0 = 0

Hw timer group 0

TIMER_GROUP_1 = 1

Hw timer group 1

TIMER_GROUP_MAX

enum **timer_idx_t**

Select a hardware timer from timer groups.

Values:

TIMER_0 = 0

Select timer0 of GROUPx

TIMER_1 = 1

Select timer1 of GROUPx

TIMER_MAX**enum timer_count_dir_t**

Decides the direction of counter.

Values:

TIMER_COUNT_DOWN = 0

Descending Count from cnt.higlcnt.low

TIMER_COUNT_UP = 1

Ascending Count from Zero

TIMER_COUNT_MAX**enum timer_start_t**

Decides whether timer is on or paused.

Values:

TIMER_PAUSE = 0

Pause timer counter

TIMER_START = 1

Start timer counter

enum timer_intr_t

Interrupt types of the timer.

Values:

TIMER_INTR_T0 = BIT(0)

interrupt of timer 0

TIMER_INTR_T1 = BIT(1)

interrupt of timer 1

TIMER_INTR_WDT = BIT(2)

interrupt of watchdog

TIMER_INTR_NONE = 0

enum timer_wdt_behavior_t

Behavior of the watchdog if a stage times out.

Values:

TIMER_WDT_OFF = 0

The stage is turned off

TIMER_WDT_INT = 1

The stage will trigger an interrupt

TIMER_WDT_RESET_CPU = 2

The stage will reset the CPU

TIMER_WDT_RESET_SYSTEM = 3

The stage will reset the whole system

enum timer_alarm_t

Decides whether to enable alarm mode.

Values:

TIMER_ALARM_DIS = 0

Disable timer alarm

TIMER_ALARM_EN = 1

Enable timer alarm

TIMER_ALARM_MAX

enum timer_intr_mode_t

Select interrupt type if running in alarm mode.

Values:

TIMER_INTR_LEVEL = 0

Interrupt mode: level mode

TIMER_INTR_MAX

enum timer_autoreload_t

Select if Alarm needs to be loaded by software or automatically reload by hardware.

Values:

TIMER_AUTORELOAD_DIS = 0

Disable auto-reload: hardware will not load counter value after an alarm event

TIMER_AUTORELOAD_EN = 1

Enable auto-reload: hardware will load counter value after an alarm event

TIMER_AUTORELOAD_MAX

2.3.20 Touch Sensor

Introduction

A touch sensor system is built on a substrate which carries electrodes and relevant connections under a protective flat surface. When a user touches the surface, the capacitance variation is used to evaluate if the touch was valid.

ESP32 can handle up to 10 capacitive touch pads / GPIOs. The sensing pads can be arranged in different combinations (e.g., matrix, slider), so that a larger area or more points can be detected. The touch pad sensing process is under the control of a hardware-implemented finite-state machine (FSM) which is initiated by software or a dedicated hardware timer.

Design, operation, and control registers of a touch sensor are discussed in [ESP32 Technical Reference Manual \(PDF\)](#). Please refer to this manual for additional details on how this subsystem works.

In-depth design details of touch sensors and firmware development guidelines for ESP32 are available in [Touch Sensor Application Note](#). If you want to test touch sensors in various configurations without building them on your own, check the [Guide for ESP32-Sense Development Kit](#).

Functionality Overview

Description of API is broken down into groups of functions to provide a quick overview of the following features:

- Initialization of touch pad driver
- Configuration of touch pad GPIO pins
- Taking measurements
- Adjusting parameters of measurements
- Filtering measurements
- Touch detection methods
- Setting up interrupts to report touch detection
- Waking up from Sleep mode on interrupt

For detailed description of a particular function, please go to Section [API Reference](#). Practical implementation of this API is covered in Section [Application Examples](#).

Initialization Before using a touch pad, you need to initialize the touch pad driver by calling the function `touch_pad_init()`. This function sets several `.._DEFAULT` driver parameters listed in [API Reference](#) under *Macros*. It also removes the information about which pads have been touched before, if any, and disables interrupts.

If the driver is not required anymore, deinitialize it by calling `touch_pad_deinit()`.

Configuration Enabling the touch sensor functionality for a particular GPIO is done with `touch_pad_config()`.

Use the function `touch_pad_set_fsm_mode()` to select if touch pad measurement (operated by FSM) should be started automatically by a hardware timer, or by software. If software mode is selected, use `touch_pad_sw_start()` to start the FSM.

Touch State Measurements The following two functions come in handy to read raw or filtered measurements from the sensor:

- `touch_pad_read()`
- `touch_pad_read_filtered()`

They can also be used, for example, to evaluate a particular touch pad design by checking the range of sensor readings when a pad is touched or released. This information can be then used to establish a touch threshold.

Note: Before using `touch_pad_read_filtered()`, you need to initialize and configure the filter by calling specific filter functions described in Section [Filtering of Measurements](#).

For the demonstration of how to use both read functions, check the application example [peripherals/touch_pad_read](#).

Optimization of Measurements A touch sensor has several configurable parameters to match the characteristics of a particular touch pad design. For instance, to sense smaller capacity changes, it is possible to narrow down the reference voltage range within which the touch pads are charged / discharged. The high and low reference voltages are set using the function `touch_pad_set_voltage()`.

Besides the ability to discern smaller capacity changes, a positive side effect is reduction of power consumption for low power applications. A likely negative effect is an increase in measurement noise. If the dynamic range of obtained readings is still satisfactory, then further reduction of power consumption might be done by reducing the measurement time with `touch_pad_set_meas_time()`.

The following list summarizes available measurement parameters and corresponding ‘set’ functions:

- Touch pad charge / discharge parameters:
 - voltage range: `touch_pad_set_voltage()`
 - speed (slope): `touch_pad_set_cnt_mode()`
- Measurement time: `touch_pad_set_meas_time()`

Relationship between the voltage range (high / low reference voltages), speed (slope), and measurement time is shown in the figure below.

The last chart *Output* represents the touch sensor reading, i.e., the count of pulses collected within the measurement time.

All functions are provided in pairs to *set* a specific parameter and to *get* the current parameter’s value, e.g., `touch_pad_set_voltage()` and `touch_pad_get_voltage()`.

Filtering of Measurements If measurements are noisy, you can filter them with provided API functions. Before using the filter, please start it by calling `touch_pad_filter_start()`.

The filter type is IIR (infinite impulse response), and it has a configurable period that can be set with the function `touch_pad_set_filter_period()`.

You can stop the filter with `touch_pad_filter_stop()`. If not required anymore, the filter can be deleted by invoking `touch_pad_filter_delete()`.

Touch Detection Touch detection is implemented in ESP32’s hardware based on the user-configured threshold and raw measurements executed by FSM. Use the functions `touch_pad_get_status()` to check which pads have been touched and `touch_pad_clear_status()` to clear the touch status information.

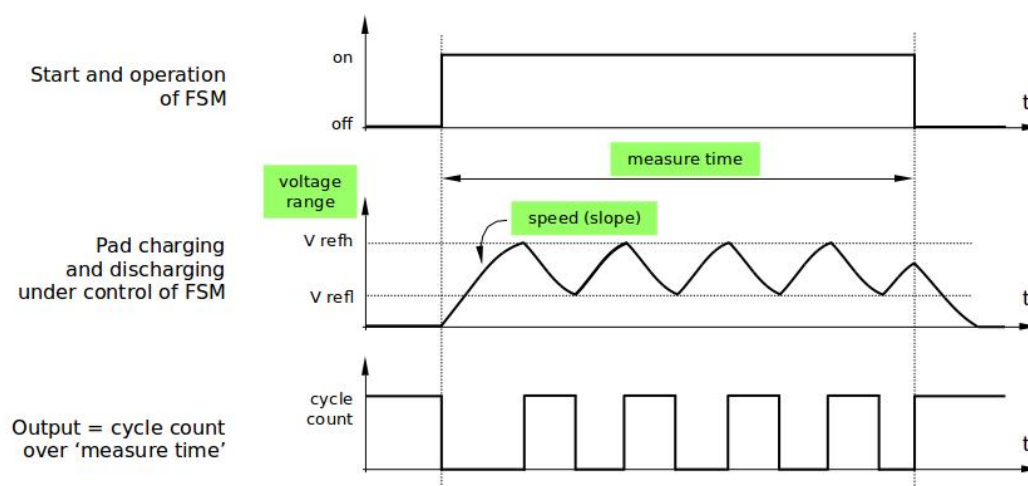


Fig. 22: Touch pad - relationship between measurement parameters

Hardware touch detection can also be wired to interrupts. This is described in the next section.

If measurements are noisy and capacity changes are small, hardware touch detection might be unreliable. To resolve this issue, instead of using hardware detection / provided interrupts, implement measurement filtering and perform touch detection in your own application. For sample implementation of both methods of touch detection, see [peripherals/touch_pad_interrupt](#).

Touch Triggered Interrupts Before enabling an interrupt on a touch detection, you should establish a touch detection threshold. Use the functions described in *Touch State Measurements* to read and display sensor measurements when a pad is touched and released. Apply a filter if measurements are noisy and relative capacity changes are small. Depending on your application and environment conditions, test the influence of temperature and power supply voltage changes on measured values.

Once a detection threshold is established, it can be set during initialization with `touch_pad_config()` or at the runtime with `touch_pad_set_thresh()`.

In the next step, configure how interrupts are triggered. They can be triggered below or above the threshold, which is set with the function `touch_pad_set_trigger_mode()`.

Finally, configure and manage interrupt calls using the following functions:

- `touch_pad_isr_register()` / `touch_pad_isr_deregister()`
- `touch_pad_intr_enable()` / `touch_pad_intr_disable()`

When interrupts are operational, you can obtain the information from which particular pad an interrupt came by invoking `touch_pad_get_status()` and clear the pad status with `touch_pad_clear_status()`.

Note: Interrupts on touch detection operate on raw / unfiltered measurements checked against user established threshold and are implemented in hardware. Enabling the software filtering API (see *Filtering of Measurements*) does not affect this process.

Wakeup from Sleep Mode If touch pad interrupts are used to wake up the chip from a sleep mode, you can select a certain configuration of pads (SET1 or both SET1 and SET2) that should be touched to trigger the interrupt and cause the subsequent wakeup. To do so, use the function `touch_pad_set_trigger_source()`.

Configuration of required bit patterns of pads may be managed for each 'SET' with:

- `touch_pad_set_group_mask () / touch_pad_get_group_mask ()`
- `touch_pad_clear_group_mask ()`

Application Examples

- Touch sensor read example: [peripherals/touch_pad_read](#).
- Touch sensor interrupt example: [peripherals/touch_pad_interrupt](#).

API Reference

Header File

- [driver/include/driver/touch_pad.h](#)

GPIO Lookup Macros Some useful macros can be used to specified the GPIO number of a touch pad channel, or vice versa. e.g.

1. `TOUCH_PAD_NUM5_GPIO_NUM` is the GPIO number of channel 5 (12);
2. `TOUCH_PAD_GPIO4_CHANNEL` is the channel number of GPIO 4 (channel 0).

Header File

- [soc/esp32/include/soc/touch_sensor_channel.h](#)

Macros

`TOUCH_PAD_GPIO4_CHANNEL`
`TOUCH_PAD_NUM0_GPIO_NUM`
`TOUCH_PAD_GPIO0_CHANNEL`
`TOUCH_PAD_NUM1_GPIO_NUM`
`TOUCH_PAD_GPIO2_CHANNEL`
`TOUCH_PAD_NUM2_GPIO_NUM`
`TOUCH_PAD_GPIO15_CHANNEL`
`TOUCH_PAD_NUM3_GPIO_NUM`
`TOUCH_PAD_GPIO13_CHANNEL`
`TOUCH_PAD_NUM4_GPIO_NUM`
`TOUCH_PAD_GPIO12_CHANNEL`
`TOUCH_PAD_NUM5_GPIO_NUM`
`TOUCH_PAD_GPIO14_CHANNEL`
`TOUCH_PAD_NUM6_GPIO_NUM`
`TOUCH_PAD_GPIO27_CHANNEL`
`TOUCH_PAD_NUM7_GPIO_NUM`
`TOUCH_PAD_GPIO33_CHANNEL`
`TOUCH_PAD_NUM8_GPIO_NUM`
`TOUCH_PAD_GPIO32_CHANNEL`
`TOUCH_PAD_NUM9_GPIO_NUM`

Header File

- [soc/include/hal/touch_sensor_types.h](#)

Macros

TOUCH_PAD_SLOPE_DEFAULT

TOUCH_PAD_TIE_OPT_DEFAULT

TOUCH_PAD_BIT_MASK_MAX

TOUCH_PAD_HIGH_VOLTAGE_THRESHOLD

TOUCH_PAD_LOW_VOLTAGE_THRESHOLD

TOUCH_PAD_ATTEN_VOLTAGE_THRESHOLD

TOUCH_PAD_INACTIVE_CONNECT_DEFAULT

TOUCH_PAD_THRESHOLD_MAX

If set touch threshold max value, The touch sensor can't be in touched status

Enumerations

enum touch_pad_t

Values:

TOUCH_PAD_NUM0 = 0

Touch pad channel 0 is GPIO4(ESP32)

TOUCH_PAD_NUM1

Touch pad channel 1 is GPIO0(ESP32) / GPIO1(ESP32-S2)

TOUCH_PAD_NUM2

Touch pad channel 2 is GPIO2(ESP32) / GPIO2(ESP32-S2)

TOUCH_PAD_NUM3

Touch pad channel 3 is GPIO15(ESP32) / GPIO3(ESP32-S2)

TOUCH_PAD_NUM4

Touch pad channel 4 is GPIO13(ESP32) / GPIO4(ESP32-S2)

TOUCH_PAD_NUM5

Touch pad channel 5 is GPIO12(ESP32) / GPIO5(ESP32-S2)

TOUCH_PAD_NUM6

Touch pad channel 6 is GPIO14(ESP32) / GPIO6(ESP32-S2)

TOUCH_PAD_NUM7

Touch pad channel 7 is GPIO27(ESP32) / GPIO7(ESP32-S2)

TOUCH_PAD_NUM8

Touch pad channel 8 is GPIO33(ESP32) / GPIO8(ESP32-S2)

TOUCH_PAD_NUM9

Touch pad channel 9 is GPIO32(ESP32) / GPIO9(ESP32-S2)

TOUCH_PAD_MAX

enum touch_high_volt_t

Values:

TOUCH_HVOLT_KEEP = -1

Touch sensor high reference voltage, no change

TOUCH_HVOLT_2V4 = 0

Touch sensor high reference voltage, 2.4V

TOUCH_HVOLT_2V5

Touch sensor high reference voltage, 2.5V

TOUCH_HVOLT_2V6

Touch sensor high reference voltage, 2.6V

TOUCH_HVOLT_2V7

Touch sensor high reference voltage, 2.7V

TOUCH_HVOLT_MAX

enum touch_low_volt_t

Values:

TOUCH_LVOLT_KEEP = -1

Touch sensor low reference voltage, no change

TOUCH_LVOLT_0V5 = 0

Touch sensor low reference voltage, 0.5V

TOUCH_LVOLT_0V6

Touch sensor low reference voltage, 0.6V

TOUCH_LVOLT_0V7

Touch sensor low reference voltage, 0.7V

TOUCH_LVOLT_0V8

Touch sensor low reference voltage, 0.8V

TOUCH_LVOLT_MAX

enum touch_volt_atten_t

Values:

TOUCH_HVOLT_ATTEN_KEEP = -1

Touch sensor high reference voltage attenuation, no change

TOUCH_HVOLT_ATTEN_1V5 = 0

Touch sensor high reference voltage attenuation, 1.5V attenuation

TOUCH_HVOLT_ATTEN_1V

Touch sensor high reference voltage attenuation, 1.0V attenuation

TOUCH_HVOLT_ATTEN_0V5

Touch sensor high reference voltage attenuation, 0.5V attenuation

TOUCH_HVOLT_ATTEN_0V

Touch sensor high reference voltage attenuation, 0V attenuation

TOUCH_HVOLT_ATTEN_MAX

enum touch_cnt_slope_t

Values:

TOUCH_PAD_SLOPE_0 = 0

Touch sensor charge / discharge speed, always zero

TOUCH_PAD_SLOPE_1 = 1

Touch sensor charge / discharge speed, slowest

TOUCH_PAD_SLOPE_2 = 2

Touch sensor charge / discharge speed

TOUCH_PAD_SLOPE_3 = 3

Touch sensor charge / discharge speed

TOUCH_PAD_SLOPE_4 = 4

Touch sensor charge / discharge speed

TOUCH_PAD_SLOPE_5 = 5

Touch sensor charge / discharge speed

TOUCH_PAD_SLOPE_6 = 6
Touch sensor charge / discharge speed

TOUCH_PAD_SLOPE_7 = 7
Touch sensor charge / discharge speed, fast

TOUCH_PAD_SLOPE_MAX

enum touch_tie_opt_t

Values:

TOUCH_PAD_TIE_OPT_LOW = 0
Initial level of charging voltage, low level

TOUCH_PAD_TIE_OPT_HIGH = 1
Initial level of charging voltage, high level

TOUCH_PAD_TIE_OPT_MAX

enum touch_fsm_mode_t

Values:

TOUCH_FSM_MODE_TIMER = 0
To start touch FSM by timer

TOUCH_FSM_MODE_SW
To start touch FSM by software trigger

TOUCH_FSM_MODE_MAX

enum touch_trigger_mode_t

Values:

TOUCH_TRIGGER_BELOW = 0
Touch interrupt will happen if counter value is less than threshold.

TOUCH_TRIGGER_ABOVE = 1
Touch interrupt will happen if counter value is larger than threshold.

TOUCH_TRIGGER_MAX

enum touch_trigger_src_t

Values:

TOUCH_TRIGGER_SOURCE_BOTH = 0
wakeup interrupt is generated if both SET1 and SET2 are “touched”

TOUCH_TRIGGER_SOURCE_SET1 = 1
wakeup interrupt is generated if SET1 is “touched”

TOUCH_TRIGGER_SOURCE_MAX

2.3.21 UART

Overview

A Universal Asynchronous Receiver/Transmitter (UART) is a hardware feature that handles communication (i.e., timing requirements and data framing) using widely-adapted asynchronous serial communication interfaces, such as RS232, RS422, RS485. A UART provides a widely adopted and cheap method to realize full-duplex or half-duplex data exchange among different devices.

The ESP32 chip has three UART controllers (UART0, UART1, and UART2) that feature an identical set of registers for ease of programming and flexibility. Each UART controller is independently configurable with parameters such as baud rate, data bit length, bit ordering, number of stop bits, parity bit etc. All the controllers are compatible with UART-enabled devices from various manufacturers and can also support Infrared Data Association protocols (IrDA).

Functional Overview

The following overview describes how to establish communication between an ESP32 and other UART devices using the functions and data types of the UART driver. The overview reflects a typical programming workflow and is broken down into the sections provided below:

1. *Setting Communication Parameters* - Setting baud rate, data bits, stop bits, etc.
2. *Setting Communication Pins* - Assigning pins for connection to a device.
3. *Driver Installation* - Allocating ESP32's resources for the UART driver.
4. *Running UART Communication* - Sending / receiving data
5. *Using Interrupts* - Triggering interrupts on specific communication events
6. *Deleting a Driver* - Freeing allocated resources if a UART communication is no longer required

Steps 1 to 3 comprise the configuration stage. Step 4 is where the UART starts operating. Steps 5 and 6 are optional.

The UART driver's functions identify each of the three UART controllers using `uart_port_t`. This identification is needed for all the following function calls.

Setting Communication Parameters UART communication parameters can be configured all in a single step or individually in multiple steps.

Single Step Call the function `uart_param_config()` and pass to it a `uart_config_t` structure. The `uart_config_t` structure should contain all the required parameters. See the example below.

```
const int uart_num = UART_NUM_2;
uart_config_t uart_config = {
    .baud_rate = 115200,
    .data_bits = UART_DATA_8_BITS,
    .parity = UART_PARITY_DISABLE,
    .stop_bits = UART_STOP_BITS_1,
    .flow_ctrl = UART_HW_FLOWCTRL_CTS_RTS,
    .rx_flow_ctrl_thresh = 122,
};
// Configure UART parameters
ESP_ERROR_CHECK(uart_param_config(uart_num, &uart_config));
```

Multiple Steps Configure specific parameters individually by calling a dedicated function from the table given below. These functions are also useful if re-configuring a single parameter.

Table 3: Functions for Configuring specific parameters individually

| Parameter to Configure | Function |
|----------------------------|---|
| Baud rate | <code>uart_set_baudrate()</code> |
| Number of transmitted bits | <code>uart_set_word_length()</code> selected out of <code>uart_word_length_t</code> |
| Parity control | <code>uart_set_parity()</code> selected out of <code>uart_parity_t</code> |
| Number of stop bits | <code>uart_set_stop_bits()</code> selected out of <code>uart_stop_bits_t</code> |
| Hardware flow control mode | <code>uart_set_hw_flow_ctrl()</code> selected out of <code>uart_hw_flowcontrol_t</code> |
| Communication mode | <code>uart_set_mode()</code> selected out of <code>uart_mode_t</code> |

Each of the above functions has a `_get_` counterpart to check the currently set value. For example, to check the current baud rate value, call `uart_get_baudrate()`.

Setting Communication Pins After setting communication parameters, configure the physical GPIO pins to which the other UART device will be connected. For this, call the function `uart_set_pin()` and specify the GPIO pin numbers to which the driver should route the Tx, Rx, RTS, and CTS signals. If you want to keep a currently allocated pin number for a specific signal, pass the macro `UART_PIN_NO_CHANGE`.

The same macro should be specified for pins that will not be used.

```
// Set UART pins(TX: IO17 (UART2 default), RX: IO16 (UART2 default), RTS: IO18,
↳CTS: IO19)
ESP_ERROR_CHECK(uart_set_pin(UART_NUM_2, UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE,
↳18, 19));
```

Driver Installation Once the communication pins are set, install the driver by calling `uart_driver_install()` and specify the following parameters:

- Size of Tx ring buffer
- Size of Rx ring buffer
- Event queue handle and size
- Flags to allocate an interrupt

The function will allocate the required ESP32 resources for the UART driver.

```
// Setup UART buffered IO with event queue
const int uart_buffer_size = (1024 * 2);
QueueHandle_t uart_queue;
// Install UART driver using an event queue here
ESP_ERROR_CHECK(uart_driver_install(UART_NUM_2, uart_buffer_size, \
uart_buffer_size, 10, &uart_queue, 0));
```

Once this step is complete, you can connect the external UART device and check the communication.

Running UART Communication Serial communication is controlled by each UART controller's finite state machine (FSM).

The process of sending data involves the following steps:

1. Write data into Tx FIFO buffer
2. FSM serializes the data
3. FSM sends the data out

The process of receiving data is similar, but the steps are reversed:

1. FSM processes an incoming serial stream and parallelizes it
2. FSM writes the data into Rx FIFO buffer
3. Read the data from Rx FIFO buffer

Therefore, an application will be limited to writing and reading data from a respective buffer using `uart_write_bytes()` and `uart_read_bytes()` respectively, and the FSM will do the rest.

Transmitting After preparing the data for transmission, call the function `uart_write_bytes()` and pass the data buffer's address and data length to it. The function will copy the data to the Tx ring buffer (either immediately or after enough space is available), and then exit. When there is free space in the Tx FIFO buffer, an interrupt service routine (ISR) moves the data from the Tx ring buffer to the Tx FIFO buffer in the background. The code below demonstrates the use of this function.

```
// Write data to UART.
char* test_str = "This is a test string.\n";
uart_write_bytes(uart_num, (const char*)test_str, strlen(test_str));
```

The function `uart_write_bytes_with_break()` is similar to `uart_write_bytes()` but adds a serial break signal at the end of the transmission. A 'serial break signal' means holding the Tx line low for a period longer than one data frame.

```
// Write data to UART, end with a break signal.
uart_write_bytes_with_break(uart_num, "test break\n", strlen("test break\n"), 100);
```


Another function for writing data to the Tx FIFO buffer is `uart_tx_chars()`. Unlike `uart_write_bytes()`, this function will not block until space is available. Instead, it will write all data which can immediately fit into the hardware Tx FIFO, and then return the number of bytes that were written.

There is a ‘companion’ function `uart_wait_tx_done()` that monitors the status of the Tx FIFO buffer and returns once it is empty.

```
// Wait for packet to be sent
const int uart_num = UART_NUM_2;
ESP_ERROR_CHECK(uart_wait_tx_done(uart_num, 100)); // wait timeout is 100 RTOS_
↳ticks (TickType_t)
```

Receiving Once the data is received by the UART and saved in the Rx FIFO buffer, it needs to be retrieved using the function `uart_read_bytes()`. Before reading data, you can check the number of bytes available in the Rx FIFO buffer by calling `uart_get_buffered_data_len()`. An example of using these functions is given below.

```
// Read data from UART.
const int uart_num = UART_NUM_2;
uint8_t data[128];
int length = 0;
ESP_ERROR_CHECK(uart_get_buffered_data_len(uart_num, (size_t*)&length));
length = uart_read_bytes(uart_num, data, length, 100);
```

If the data in the Rx FIFO buffer is no longer needed, you can clear the buffer by calling `uart_flush()`.

Software Flow Control If the hardware flow control is disabled, you can manually set the RTS and DTR signal levels by using the functions `uart_set_rts()` and `uart_set_dtr()` respectively.

Communication Mode Selection The UART controller supports a number of communication modes. A mode can be selected using the function `uart_set_mode()`. Once a specific mode is selected, the UART driver will handle the behavior of a connected UART device accordingly. As an example, it can control the RS485 driver chip using the RTS line to allow half-duplex RS485 communication.

```
// Setup UART in rs485 half duplex mode
ESP_ERROR_CHECK(uart_set_mode(uart_num, UART_MODE_RS485_HALF_DUPLEX));
```

Using Interrupts There are many interrupts that can be generated following specific UART states or detected errors. The full list of available interrupts is provided in [ESP32 Technical Reference Manual \(PDF\)](#). You can enable or disable specific interrupts by calling `uart_enable_intr_mask()` or `uart_disable_intr_mask()` respectively. The mask of all interrupts is available as `UART_INTR_MASK`.

By default, the `uart_driver_install()` function installs the driver’s internal interrupt handler to manage the Tx and Rx ring buffers and provides high-level API functions like events (see below). It is also possible to register a lower level interrupt handler instead using `uart_isr_register()`, and to free it again using `uart_isr_free()`. Some UART driver functions which use the Tx and Rx ring buffers, events, etc. will not automatically work in this case - it is necessary to handle the interrupts directly in the ISR. Inside the custom handler implementation, clear the interrupt status bits using `uart_clear_intr_status()`.

The API provides a convenient way to handle specific interrupts discussed in this document by wrapping them into dedicated functions:

- **Event detection:** There are several events defined in `uart_event_type_t` that may be reported to a user application using the FreeRTOS queue functionality. You can enable this functionality when calling `uart_driver_install()` described in [Driver Installation](#). An example of using Event detection can be found in [peripherals/uart_events](#).

- **FIFO space threshold or transmission timeout reached:** The Tx and Rx FIFO buffers can trigger an interrupt when they are filled with a specific number of characters, or on a timeout of sending or receiving data. To use these interrupts, do the following:
 - Configure respective threshold values of the buffer length and timeout by entering them in the structure `uart_intr_config_t` and calling `uart_intr_config()`
 - Enable the interrupts using the functions `uart_enable_tx_intr()` and `uart_enable_rx_intr()`
 - Disable these interrupts using the corresponding functions `uart_disable_tx_intr()` or `uart_disable_rx_intr()`
- **Pattern detection:** An interrupt triggered on detecting a ‘pattern’ of the same character being received/sent repeatedly for a number of times. This functionality is demonstrated in the example [peripherals/uart/uart_events](#). It can be used, e.g., to detect a command string followed by a specific number of identical characters (the ‘pattern’) added at the end of the command string. The following functions are available:
 - Configure and enable this interrupt using `uart_enable_pattern_det_intr()`
 - Disable the interrupt using `uart_disable_pattern_det_intr()`

Macros The API also defines several macros. For example, `UART_FIFO_LEN` defines the length of hardware FIFO buffers; `UART_BITRATE_MAX` gives the maximum baud rate supported by the UART controllers, etc.

Deleting a Driver If the communication established with `uart_driver_install()` is no longer required, the driver can be removed to free allocated resources by calling `uart_driver_delete()`.

Overview of RS485 specific communication options

Note: The following section will use `[UART_REGISTER_NAME].[UART_FIELD_BIT]` to refer to UART register fields/bits. To find more information on a specific option bit, open [Register Summary in the ESP32 Technical Reference Manual](#) (PDF), use the register name to navigate to the register description and then find the field/bit.

- `UART_RS485_CONF_REG.UART_RS485_EN`: setting this bit enables RS485 communication mode support.
- `UART_RS485_CONF_REG.UART_RS485TX_RX_EN`: if this bit is set, the transmitter’s output signal loops back to the receiver’s input signal.
- `UART_RS485_CONF_REG.UART_RS485RXBY_TX_EN`: if this bit is set, the transmitter will still be sending data if the receiver is busy (remove collisions automatically by hardware).

The ESP32’s RS485 UART hardware can detect signal collisions during transmission of a datagram and generate the interrupt `UART_RS485_CLASH_INT` if this interrupt is enabled. The term collision means that a transmitted datagram is not equal to the one received on the other end. Data collisions are usually associated with the presence of other active devices on the bus or might occur due to bus errors.

The collision detection feature allows handling collisions when their interrupts are activated and triggered. The interrupts `UART_RS485_FRM_ERR_INT` and `UART_RS485_PARITY_ERR_INT` can be used with the collision detection feature to control frame errors and parity bit errors accordingly in RS485 mode. This functionality is supported in the UART driver and can be used by selecting the `UART_MODE_RS485_APP_CTRL` mode (see the function `uart_set_mode()`).

The collision detection feature can work with circuit A and circuit C (see Section [Interface Connection Options](#)). In the case of using circuit A or B, the RTS pin connected to the DE pin of the bus driver should be controlled by the user application. Use the function `uart_get_collision_flag()` to check if the collision detection flag has been raised.

The ESP32’s UART controllers themselves do not support half-duplex communication as they cannot provide automatic control of the RTS pin connected to the `~RE/DE` input of RS485 bus driver. However, half-duplex communication can be achieved via software control of the RTS pin by the UART driver. This can be enabled by selecting the `UART_MODE_RS485_HALF_DUPLEX` mode when calling `uart_set_mode()`.

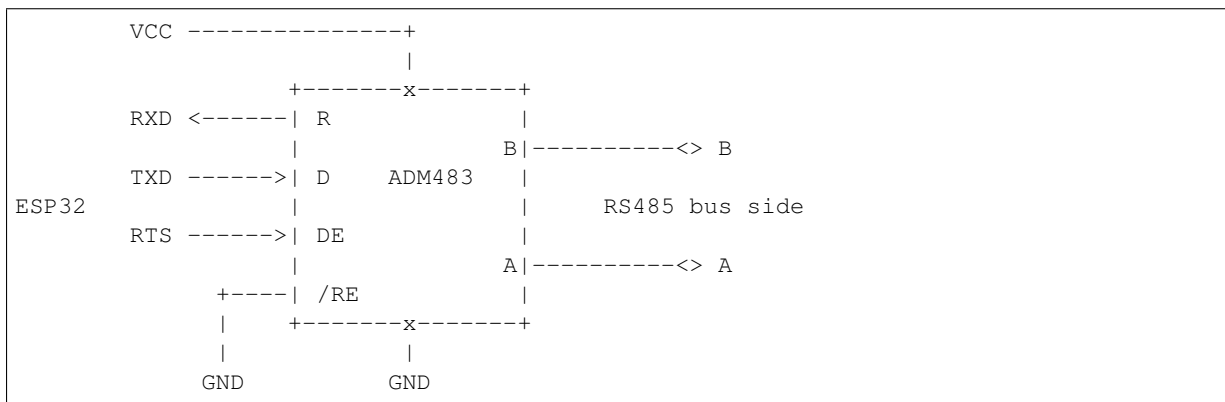
Once the host starts writing data to the Tx FIFO buffer, the UART driver automatically asserts the RTS pin (logic 1); once the last bit of the data has been transmitted, the driver de-asserts the RTS pin (logic 0). To use this mode, the software would have to disable the hardware flow control function. This mode works with all the used circuits shown below.

Interface Connection Options This section provides example schematics to demonstrate the basic aspects of ESP32's RS485 interface connection.

Note:

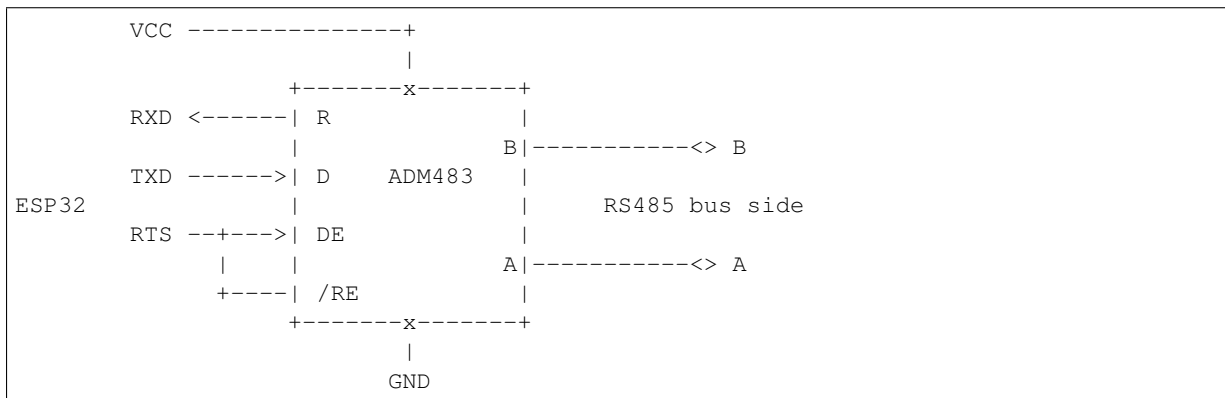
- The schematics below do **not** necessarily contain **all required elements**.
- The **analog devices** ADM483 & ADM2483 are examples of common RS485 transceivers and **can be replaced** with other similar transceivers.

Circuit A: Collision Detection Circuit



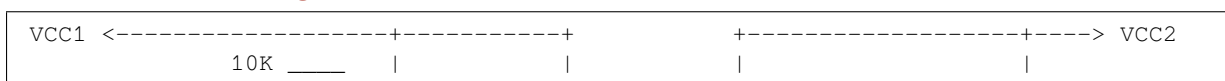
This circuit is preferable because it allows for collision detection and is quite simple at the same time. The receiver in the line driver is constantly enabled, which allows the UART to monitor the RS485 bus. Echo suppression is performed by the ESP32 hardware when the bit `UART_RS485_CONF_REG.UART_RS485TX_RX_EN` is enabled.

Circuit B: Manual Switching Transmitter/Receiver Without Collision Detection



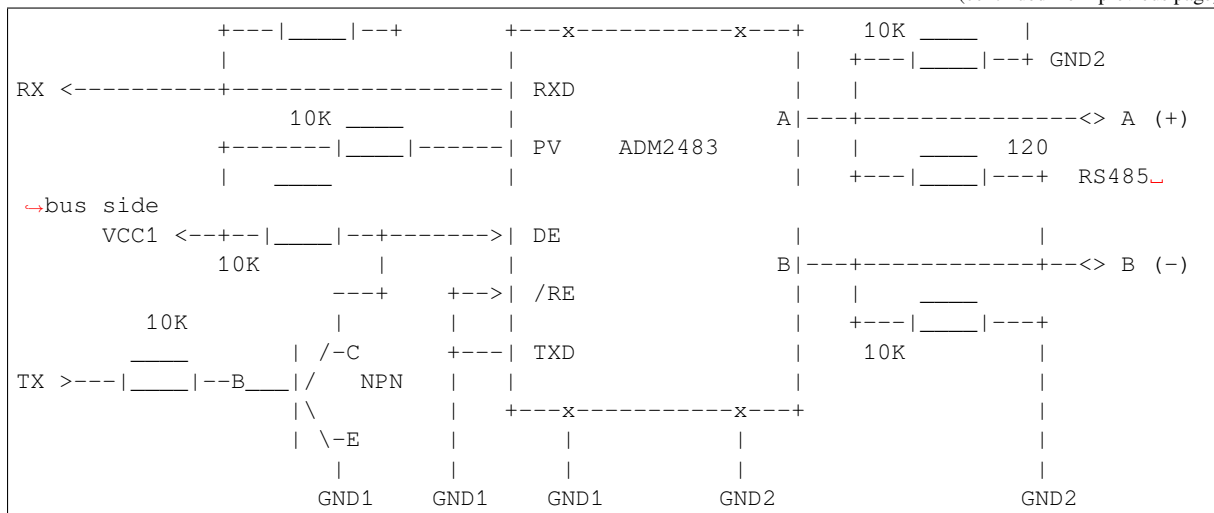
This circuit does not allow for collision detection. It suppresses the null bytes that the hardware receives when the bit `UART_RS485_CONF_REG.UART_RS485TX_RX_EN` is set. The bit `UART_RS485_CONF_REG.UART_RS485RXBY_TX_EN` is not applicable in this case.

Circuit C: Auto Switching Transmitter/Receiver



(continues on next page)

(continued from previous page)



This galvanically isolated circuit does not require RTS pin control by a software application or driver because it controls the transceiver direction automatically. However, it requires suppressing null bytes during transmission by setting `UART_RS485_CONF_REG.UART_RS485RXBY_TX_EN` to 1 and `UART_RS485_CONF_REG.UART_RS485TX_RX_EN` to 0. This setup can work in any RS485 UART mode or even in `UART_MODE_UART`.

Application Examples

The table below describes the code examples available in the directory [peripherals/uart/](#).

| Code Example | Description |
|---|---|
| peripherals/uart/uart_echo | Configuring UART settings, installing the UART driver, and reading/writing over the UART1 interface. |
| peripherals/uart/uart_events | Reporting various communication events, using pattern detection interrupts. |
| peripherals/uart/uart_async_rxtxtasks | Transmitting and receiving data in two separate FreeRTOS tasks over the same UART. |
| peripherals/uart/uart_select | Using synchronous I/O multiplexing for UART file descriptors. |
| peripherals/uart/uart_echo_rs485 | Setting up UART driver to communicate over RS485 interface in half-duplex mode. This example is similar to peripherals/uart/uart_echo but allows communication through an RS485 interface chip connected to ESP32 pins. |
| peripherals/uart/nmea0183_parser | Obtaining GPS information by parsing NMEA0183 statements received from GPS via the UART peripheral. |

API Reference

Header File

- [driver/include/driver/uart.h](#)

Functions

`esp_err_t uart_driver_install(uart_port_t uart_num, int rx_buffer_size, int tx_buffer_size, int queue_size, QueueHandle_t *uart_queue, int intr_alloc_flags)`

Install UART driver and set the UART to the default configuration.

UART ISR handler will be attached to the same CPU core that this function is running on.

Note `Rx_buffer_size` should be greater than `UART_FIFO_LEN`. `Tx_buffer_size` should be either zero or greater than `UART_FIFO_LEN`.

Return

- ESP_OK Success
- ESP_FAIL Parameter error

Parameters

- `uart_num`: UART port number, the max port number is (UART_NUM_MAX -1).
- `rx_buffer_size`: UART RX ring buffer size.
- `tx_buffer_size`: UART TX ring buffer size. If set to zero, driver will not use TX buffer, TX function will block task until all data have been sent out.
- `queue_size`: UART event queue size/depth.
- `uart_queue`: UART event queue handle (out param). On success, a new queue handle is written here to provide access to UART events. If set to NULL, driver will not use an event queue.
- `intr_alloc_flags`: Flags used to allocate the interrupt. One or multiple (ORred) ESP_INTR_FLAG_* values. See `esp_intr_alloc.h` for more info. Do not set ESP_INTR_FLAG_IRAM here (the driver's ISR handler is not located in IRAM)

esp_err_t **uart_driver_delete** (*uart_port_t* *uart_num*)

Uninstall UART driver.

Return

- ESP_OK Success
- ESP_FAIL Parameter error

Parameters

- `uart_num`: UART port number, the max port number is (UART_NUM_MAX -1).

bool **uart_is_driver_installed** (*uart_port_t* *uart_num*)

Checks whether the driver is installed or not.

Return

- true driver is installed
- false driver is not installed

Parameters

- `uart_num`: UART port number, the max port number is (UART_NUM_MAX -1).

esp_err_t **uart_set_word_length** (*uart_port_t* *uart_num*, *uart_word_length_t* *data_bit*)

Set UART data bits.

Return

- ESP_OK Success
- ESP_FAIL Parameter error

Parameters

- `uart_num`: UART port number, the max port number is (UART_NUM_MAX -1).
- `data_bit`: UART data bits

esp_err_t **uart_get_word_length** (*uart_port_t* *uart_num*, *uart_word_length_t* **data_bit*)

Get the UART data bit configuration.

Return

- ESP_FAIL Parameter error
- ESP_OK Success, result will be put in (*data_bit)

Parameters

- `uart_num`: UART port number, the max port number is (UART_NUM_MAX -1).
- `data_bit`: Pointer to accept value of UART data bits.

esp_err_t **uart_set_stop_bits** (*uart_port_t* *uart_num*, *uart_stop_bits_t* *stop_bits*)

Set UART stop bits.

Return

- ESP_OK Success
- ESP_FAIL Fail

Parameters

- `uart_num`: UART port number, the max port number is (UART_NUM_MAX -1).
- `stop_bits`: UART stop bits

esp_err_t **uart_get_stop_bits** (*uart_port_t* *uart_num*, *uart_stop_bits_t* **stop_bits*)

Get the UART stop bit configuration.

Return

- ESP_FAIL Parameter error
- ESP_OK Success, result will be put in (*stop_bit)

Parameters

- *uart_num*: UART port number, the max port number is (UART_NUM_MAX -1).
- *stop_bits*: Pointer to accept value of UART stop bits.

esp_err_t **uart_set_parity** (*uart_port_t* *uart_num*, *uart_parity_t* *parity_mode*)

Set UART parity mode.

Return

- ESP_FAIL Parameter error
- ESP_OK Success

Parameters

- *uart_num*: UART port number, the max port number is (UART_NUM_MAX -1).
- *parity_mode*: the enum of uart parity configuration

esp_err_t **uart_get_parity** (*uart_port_t* *uart_num*, *uart_parity_t* **parity_mode*)

Get the UART parity mode configuration.

Return

- ESP_FAIL Parameter error
- ESP_OK Success, result will be put in (*parity_mode)

Parameters

- *uart_num*: UART port number, the max port number is (UART_NUM_MAX -1).
- *parity_mode*: Pointer to accept value of UART parity mode.

esp_err_t **uart_set_baudrate** (*uart_port_t* *uart_num*, *uint32_t* *baudrate*)

Set UART baud rate.

Return

- ESP_FAIL Parameter error
- ESP_OK Success

Parameters

- *uart_num*: UART port number, the max port number is (UART_NUM_MAX -1).
- *baudrate*: UART baud rate.

esp_err_t **uart_get_baudrate** (*uart_port_t* *uart_num*, *uint32_t* **baudrate*)

Get the UART baud rate configuration.

Return

- ESP_FAIL Parameter error
- ESP_OK Success, result will be put in (*baudrate)

Parameters

- *uart_num*: UART port number, the max port number is (UART_NUM_MAX -1).
- *baudrate*: Pointer to accept value of UART baud rate

esp_err_t **uart_set_line_inverse** (*uart_port_t* *uart_num*, *uint32_t* *inverse_mask*)

Set UART line inverse mode.

Return

- ESP_OK Success
- ESP_FAIL Parameter error

Parameters

- *uart_num*: UART port number, the max port number is (UART_NUM_MAX -1).
- *inverse_mask*: Choose the wires that need to be inverted. Using the ORred mask of *uart_signal_inv_t*

esp_err_t **uart_set_hw_flow_ctrl** (*uart_port_t* *uart_num*, *uart_hw_flowcontrol_t* *flow_ctrl*, *uint8_t* *rx_thresh*)

Set hardware flow control.

Return

- ESP_OK Success
- ESP_FAIL Parameter error

Parameters

- `uart_num`: UART port number, the max port number is (UART_NUM_MAX -1).
- `flow_ctrl`: Hardware flow control mode
- `rx_thresh`: Threshold of Hardware RX flow control (0 ~ UART_FIFO_LEN). Only when UART_HW_FLOWCTRL_RTS is set, will the `rx_thresh` value be set.

esp_err_t **uart_set_sw_flow_ctrl** (*uart_port_t* `uart_num`, bool `enable`, uint8_t `rx_thresh_xon`, uint8_t `rx_thresh_xoff`)

Set software flow control.

Return

- ESP_OK Success
- ESP_FAIL Parameter error

Parameters

- `uart_num`: UART_NUM_0, UART_NUM_1 or UART_NUM_2
- `enable`: switch on or off
- `rx_thresh_xon`: low water mark
- `rx_thresh_xoff`: high water mark

esp_err_t **uart_get_hw_flow_ctrl** (*uart_port_t* `uart_num`, *uart_hw_flowcontrol_t* `*flow_ctrl`)

Get the UART hardware flow control configuration.

Return

- ESP_FAIL Parameter error
- ESP_OK Success, result will be put in (`*flow_ctrl`)

Parameters

- `uart_num`: UART port number, the max port number is (UART_NUM_MAX -1).
- `flow_ctrl`: Option for different flow control mode.

esp_err_t **uart_clear_intr_status** (*uart_port_t* `uart_num`, uint32_t `clr_mask`)

Clear UART interrupt status.

Return

- ESP_OK Success
- ESP_FAIL Parameter error

Parameters

- `uart_num`: UART port number, the max port number is (UART_NUM_MAX -1).
- `clr_mask`: Bit mask of the interrupt status to be cleared.

esp_err_t **uart_enable_intr_mask** (*uart_port_t* `uart_num`, uint32_t `enable_mask`)

Set UART interrupt enable.

Return

- ESP_OK Success
- ESP_FAIL Parameter error

Parameters

- `uart_num`: UART port number, the max port number is (UART_NUM_MAX -1).
- `enable_mask`: Bit mask of the enable bits.

esp_err_t **uart_disable_intr_mask** (*uart_port_t* `uart_num`, uint32_t `disable_mask`)

Clear UART interrupt enable bits.

Return

- ESP_OK Success
- ESP_FAIL Parameter error

Parameters

- `uart_num`: UART port number, the max port number is (UART_NUM_MAX -1).
- `disable_mask`: Bit mask of the disable bits.

esp_err_t **uart_enable_rx_intr** (*uart_port_t* `uart_num`)

Enable UART RX interrupt (RX_FULL & RX_TIMEOUT INTERRUPT)

Return

- ESP_OK Success
- ESP_FAIL Parameter error

Parameters

- `uart_num`: UART port number, the max port number is (UART_NUM_MAX -1).

esp_err_t **uart_disable_rx_intr** (*uart_port_t* *uart_num*)

Disable UART RX interrupt (RX_FULL & RX_TIMEOUT INTERRUPT)

Return

- ESP_OK Success
- ESP_FAIL Parameter error

Parameters

- `uart_num`: UART port number, the max port number is (UART_NUM_MAX -1).

esp_err_t **uart_disable_tx_intr** (*uart_port_t* *uart_num*)

Disable UART TX interrupt (TX_FULL & TX_TIMEOUT INTERRUPT)

Return

- ESP_OK Success
- ESP_FAIL Parameter error

Parameters

- `uart_num`: UART port number

esp_err_t **uart_enable_tx_intr** (*uart_port_t* *uart_num*, int *enable*, int *thresh*)

Enable UART TX interrupt (TX_FULL & TX_TIMEOUT INTERRUPT)

Return

- ESP_OK Success
- ESP_FAIL Parameter error

Parameters

- `uart_num`: UART port number, the max port number is (UART_NUM_MAX -1).
- `enable`: 1: enable; 0: disable
- `thresh`: Threshold of TX interrupt, 0 ~ UART_FIFO_LEN

esp_err_t **uart_isr_register** (*uart_port_t* *uart_num*, void (**fn*)) void *

, void **arg*, int *intr_alloc_flags*, *uart_isr_handle_t* **handle* Register UART interrupt handler (ISR).

Note UART ISR handler will be attached to the same CPU core that this function is running on.

Return

- ESP_OK Success
- ESP_FAIL Parameter error

Parameters

- `uart_num`: UART port number, the max port number is (UART_NUM_MAX -1).
- `fn`: Interrupt handler function.
- `arg`: parameter for handler function
- `intr_alloc_flags`: Flags used to allocate the interrupt. One or multiple (ORred) ESP_INTR_FLAG_* values. See `esp_intr_alloc.h` for more info.
- `handle`: Pointer to return handle. If non-NULL, a handle for the interrupt will be returned here.

esp_err_t **uart_isr_free** (*uart_port_t* *uart_num*)

Free UART interrupt handler registered by `uart_isr_register`. Must be called on the same core as `uart_isr_register` was called.

Return

- ESP_OK Success
- ESP_FAIL Parameter error

Parameters

- `uart_num`: UART port number, the max port number is (UART_NUM_MAX -1).

esp_err_t **uart_set_pin** (*uart_port_t* *uart_num*, int *tx_io_num*, int *rx_io_num*, int *rts_io_num*, int *cts_io_num*)

Set UART pin number.

Note Internal signal can be output to multiple GPIO pads. Only one GPIO pad can connect with input signal.

Note Instead of GPIO number a macro 'UART_PIN_NO_CHANGE' may be provided to keep the currently allocated pin.

Return

- ESP_OK Success
- ESP_FAIL Parameter error

Parameters

- `uart_num`: UART port number, the max port number is (UART_NUM_MAX -1).
- `tx_io_num`: UART TX pin GPIO number.
- `rx_io_num`: UART RX pin GPIO number.
- `rts_io_num`: UART RTS pin GPIO number.
- `cts_io_num`: UART CTS pin GPIO number.

esp_err_t **uart_set_rts** (*uart_port_t* `uart_num`, int `level`)

Manually set the UART RTS pin level.

Note UART must be configured with hardware flow control disabled.

Return

- ESP_OK Success
- ESP_FAIL Parameter error

Parameters

- `uart_num`: UART port number, the max port number is (UART_NUM_MAX -1).
- `level`: 1: RTS output low (active); 0: RTS output high (block)

esp_err_t **uart_set_dtr** (*uart_port_t* `uart_num`, int `level`)

Manually set the UART DTR pin level.

Return

- ESP_OK Success
- ESP_FAIL Parameter error

Parameters

- `uart_num`: UART port number, the max port number is (UART_NUM_MAX -1).
- `level`: 1: DTR output low; 0: DTR output high

esp_err_t **uart_set_tx_idle_num** (*uart_port_t* `uart_num`, uint16_t `idle_num`)

Set UART idle interval after tx FIFO is empty.

Return

- ESP_OK Success
- ESP_FAIL Parameter error

Parameters

- `uart_num`: UART port number, the max port number is (UART_NUM_MAX -1).
- `idle_num`: idle interval after tx FIFO is empty(unit: the time it takes to send one bit under current baudrate)

esp_err_t **uart_param_config** (*uart_port_t* `uart_num`, const *uart_config_t* *`uart_config`)

Set UART configuration parameters.

Return

- ESP_OK Success
- ESP_FAIL Parameter error

Parameters

- `uart_num`: UART port number, the max port number is (UART_NUM_MAX -1).
- `uart_config`: UART parameter settings

esp_err_t **uart_intr_config** (*uart_port_t* `uart_num`, const *uart_intr_config_t* *`intr_conf`)

Configure UART interrupts.

Return

- ESP_OK Success
- ESP_FAIL Parameter error

Parameters

- `uart_num`: UART port number, the max port number is (UART_NUM_MAX -1).

- `intr_conf`: UART interrupt settings

`esp_err_t uart_wait_tx_done (uart_port_t uart_num, TickType_t ticks_to_wait)`

Wait until UART TX FIFO is empty.

Return

- `ESP_OK` Success
- `ESP_FAIL` Parameter error
- `ESP_ERR_TIMEOUT` Timeout

Parameters

- `uart_num`: UART port number, the max port number is (`UART_NUM_MAX - 1`).
- `ticks_to_wait`: Timeout, count in RTOS ticks

`int uart_tx_chars (uart_port_t uart_num, const char *buffer, uint32_t len)`

Send data to the UART port from a given buffer and length.

This function will not wait for enough space in TX FIFO. It will just fill the available TX FIFO and return when the FIFO is full.

Note This function should only be used when UART TX buffer is not enabled.

Return

- (-1) Parameter error
- `OTHERS (>=0)` The number of bytes pushed to the TX FIFO

Parameters

- `uart_num`: UART port number, the max port number is (`UART_NUM_MAX - 1`).
- `buffer`: data buffer address
- `len`: data length to send

`int uart_write_bytes (uart_port_t uart_num, const char *src, size_t size)`

Send data to the UART port from a given buffer and length,.

If the UART driver' s parameter 'tx_buffer_size' is set to zero: This function will not return until all the data have been sent out, or at least pushed into TX FIFO.

Otherwise, if the 'tx_buffer_size' > 0, this function will return after copying all the data to tx ring buffer, UART ISR will then move data from the ring buffer to TX FIFO gradually.

Return

- (-1) Parameter error
- `OTHERS (>=0)` The number of bytes pushed to the TX FIFO

Parameters

- `uart_num`: UART port number, the max port number is (`UART_NUM_MAX - 1`).
- `src`: data buffer address
- `size`: data length to send

`int uart_write_bytes_with_break (uart_port_t uart_num, const char *src, size_t size, int brk_len)`

Send data to the UART port from a given buffer and length,.

If the UART driver' s parameter 'tx_buffer_size' is set to zero: This function will not return until all the data and the break signal have been sent out. After all data is sent out, send a break signal.

Otherwise, if the 'tx_buffer_size' > 0, this function will return after copying all the data to tx ring buffer, UART ISR will then move data from the ring buffer to TX FIFO gradually. After all data sent out, send a break signal.

Return

- (-1) Parameter error
- `OTHERS (>=0)` The number of bytes pushed to the TX FIFO

Parameters

- `uart_num`: UART port number, the max port number is (`UART_NUM_MAX - 1`).
- `src`: data buffer address
- `size`: data length to send
- `brk_len`: break signal duration(unit: the time it takes to send one bit at current baudrate)

`int uart_read_bytes (uart_port_t uart_num, uint8_t *buf, uint32_t length, TickType_t ticks_to_wait)`
UART read bytes from UART buffer.

Return

- (-1) Error
- OTHERS (>=0) The number of bytes read from UART FIFO

Parameters

- `uart_num`: UART port number, the max port number is (UART_NUM_MAX -1).
- `buf`: pointer to the buffer.
- `length`: data length
- `ticks_to_wait`: sTimeout, count in RTOS ticks

`esp_err_t uart_flush (uart_port_t uart_num)`

Alias of `uart_flush_input`. UART ring buffer flush. This will discard all data in the UART RX buffer.

Note Instead of waiting the data sent out, this function will clear UART rx buffer. In order to send all the data in tx FIFO, we can use `uart_wait_tx_done` function.

Return

- ESP_OK Success
- ESP_FAIL Parameter error

Parameters

- `uart_num`: UART port number, the max port number is (UART_NUM_MAX -1).

`esp_err_t uart_flush_input (uart_port_t uart_num)`

Clear input buffer, discard all the data is in the ring-buffer.

Note In order to send all the data in tx FIFO, we can use `uart_wait_tx_done` function.

Return

- ESP_OK Success
- ESP_FAIL Parameter error

Parameters

- `uart_num`: UART port number, the max port number is (UART_NUM_MAX -1).

`esp_err_t uart_get_buffered_data_len (uart_port_t uart_num, size_t *size)`

UART get RX ring buffer cached data length.

Return

- ESP_OK Success
- ESP_FAIL Parameter error

Parameters

- `uart_num`: UART port number, the max port number is (UART_NUM_MAX -1).
- `size`: Pointer of `size_t` to accept cached data length

`esp_err_t uart_disable_pattern_det_intr (uart_port_t uart_num)`

UART disable pattern detect function. Designed for applications like ‘AT commands’. When the hardware detects a series of one same character, the interrupt will be triggered.

Return

- ESP_OK Success
- ESP_FAIL Parameter error

Parameters

- `uart_num`: UART port number, the max port number is (UART_NUM_MAX -1).

`esp_err_t uart_enable_pattern_det_baud_intr (uart_port_t uart_num, char pattern_chr, uint8_t chr_num, int chr_tout, int post_idle, int pre_idle)`

UART enable pattern detect function. Designed for applications like ‘AT commands’. When the hardware detect a series of one same character, the interrupt will be triggered.

Return

- ESP_OK Success
- ESP_FAIL Parameter error

Parameters

- `uart_num`: UART port number.

- `pattern_chr`: character of the pattern.
- `chr_num`: number of the character, 8bit value.
- `chr_tout`: timeout of the interval between each pattern characters, 16bit value, unit is the baud-rate cycle you configured. When the duration is more than this value, it will not take this data as `at_cmd` char.
- `post_idle`: idle time after the last pattern character, 16bit value, unit is the baud-rate cycle you configured. When the duration is less than this value, it will not take the previous data as the last `at_cmd` char
- `pre_idle`: idle time before the first pattern character, 16bit value, unit is the baud-rate cycle you configured. When the duration is less than this value, it will not take this data as the first `at_cmd` char.

int `uart_pattern_pop_pos` (*uart_port_t* *uart_num*)

Return the nearest detected pattern position in buffer. The positions of the detected pattern are saved in a queue, this function will dequeue the first pattern position and move the pointer to next pattern position.

The following APIs will modify the pattern position info: `uart_flush_input`, `uart_read_bytes`, `uart_driver_delete`, `uart_pop_pattern_pos` It is the application's responsibility to ensure atomic access to the pattern queue and the rx data buffer when using pattern detect feature.

Note If the RX buffer is full and flow control is not enabled, the detected pattern may not be found in the rx buffer due to overflow.

Return

- (-1) No pattern found for current index or parameter error
- others the pattern position in rx buffer.

Parameters

- `uart_num`: UART port number, the max port number is (UART_NUM_MAX -1).

int `uart_pattern_get_pos` (*uart_port_t* *uart_num*)

Return the nearest detected pattern position in buffer. The positions of the detected pattern are saved in a queue, This function do nothing to the queue.

The following APIs will modify the pattern position info: `uart_flush_input`, `uart_read_bytes`, `uart_driver_delete`, `uart_pop_pattern_pos` It is the application's responsibility to ensure atomic access to the pattern queue and the rx data buffer when using pattern detect feature.

Note If the RX buffer is full and flow control is not enabled, the detected pattern may not be found in the rx buffer due to overflow.

Return

- (-1) No pattern found for current index or parameter error
- others the pattern position in rx buffer.

Parameters

- `uart_num`: UART port number, the max port number is (UART_NUM_MAX -1).

esp_err_t `uart_pattern_queue_reset` (*uart_port_t* *uart_num*, int *queue_length*)

Allocate a new memory with the given length to save record the detected pattern position in rx buffer.

Return

- `ESP_ERR_NO_MEM` No enough memory
- `ESP_ERR_INVALID_STATE` Driver not installed
- `ESP_FAIL` Parameter error
- `ESP_OK` Success

Parameters

- `uart_num`: UART port number, the max port number is (UART_NUM_MAX -1).
- `queue_length`: Max queue length for the detected pattern. If the queue length is not large enough, some pattern positions might be lost. Set this value to the maximum number of patterns that could be saved in data buffer at the same time.

esp_err_t `uart_set_mode` (*uart_port_t* *uart_num*, *uart_mode_t* *mode*)

UART set communication mode.

Note This function must be executed after `uart_driver_install()`, when the driver object is initialized.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `uart_num`: Uart number to configure, the max port number is (UART_NUM_MAX -1).
- `mode`: UART UART mode to set

esp_err_t **uart_set_rx_full_threshold** (*uart_port_t* *uart_num*, int *threshold*)

Set uart threshold value for RX fifo full.

Note If application is using higher baudrate and it is observed that bytes in hardware RX fifo are overwritten then this threshold can be reduced

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_INVALID_STATE Driver is not installed

Parameters

- `uart_num`: UART_NUM_0, UART_NUM_1 or UART_NUM_2
- `threshold`: Threshold value above which RX fifo full interrupt is generated

esp_err_t **uart_set_tx_empty_threshold** (*uart_port_t* *uart_num*, int *threshold*)

Set uart threshold values for TX fifo empty.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_INVALID_STATE Driver is not installed

Parameters

- `uart_num`: UART_NUM_0, UART_NUM_1 or UART_NUM_2
- `threshold`: Threshold value below which TX fifo empty interrupt is generated

esp_err_t **uart_set_rx_timeout** (*uart_port_t* *uart_num*, const uint8_t *tout_thresh*)

UART set threshold timeout for TOUT feature.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_INVALID_STATE Driver is not installed

Parameters

- `uart_num`: Uart number to configure, the max port number is (UART_NUM_MAX -1).
- `tout_thresh`: This parameter defines timeout threshold in uart symbol periods. The maximum value of threshold is 126. `tout_thresh = 1`, defines TOUT interrupt timeout equal to transmission time of one symbol (~11 bit) on current baudrate. If the time is expired the UART_RXFIFO_TOUT_INT interrupt is triggered. If `tout_thresh == 0`, the TOUT feature is disabled.

esp_err_t **uart_get_collision_flag** (*uart_port_t* *uart_num*, bool **collision_flag*)

Returns collision detection flag for RS485 mode Function returns the collision detection flag into variable pointed by `collision_flag`. `*collision_flag = true`, if collision detected else it is equal to false. This function should be executed when actual transmission is completed (after `uart_write_bytes()`).

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

Parameters

- `uart_num`: Uart number to configure the max port number is (UART_NUM_MAX -1).
- `collision_flag`: Pointer to variable of type bool to return collision flag.

esp_err_t **uart_set_wakeup_threshold** (*uart_port_t* *uart_num*, int *wakeup_threshold*)

Set the number of RX pin signal edges for light sleep wakeup.

UART can be used to wake up the system from light sleep. This feature works by counting the number of positive edges on RX pin and comparing the count to the threshold. When the count exceeds the threshold, system is woken up from light sleep. This function allows setting the threshold value.

Stop bit and parity bits (if enabled) also contribute to the number of edges. For example, letter ‘a’ with ASCII code 97 is encoded as 0100001101 on the wire (with 8n1 configuration), start and stop bits included. This sequence has 3 positive edges (transitions from 0 to 1). Therefore, to wake up the system when ‘a’ is sent, set `wakeup_threshold=3`.

The character that triggers wakeup is not received by UART (i.e. it can not be obtained from UART FIFO). Depending on the baud rate, a few characters after that will also not be received. Note that when the chip enters and exits light sleep mode, APB frequency will be changing. To make sure that UART has correct baud rate all the time, select REF_TICK as UART clock source, by setting `use_ref_tick` field in `uart_config_t` to true.

Note in ESP32, the wakeup signal can only be input via IO_MUX (i.e. GPIO3 should be configured as function_1 to wake up UART0, GPIO9 should be configured as function_5 to wake up UART1), UART2 does not support light sleep wakeup feature.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if `uart_num` is incorrect or `wakeup_threshold` is outside of [3, 0x3ff] range.

Parameters

- `uart_num`: UART number, the max port number is (UART_NUM_MAX -1).
- `wakeup_threshold`: number of RX edges for light sleep wakeup, value is 3 .. 0x3ff.

`esp_err_t uart_get_wakeup_threshold (uart_port_t uart_num, int *out_wakeup_threshold)`

Get the number of RX pin signal edges for light sleep wakeup.

See description of `uart_set_wakeup_threshold` for the explanation of UART wakeup feature.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if `out_wakeup_threshold` is NULL

Parameters

- `uart_num`: UART number, the max port number is (UART_NUM_MAX -1).
- [out] `out_wakeup_threshold`: output, set to the current value of wakeup threshold for the given UART.

`esp_err_t uart_wait_tx_idle_polling (uart_port_t uart_num)`

Wait until UART tx memory empty and the last char send ok (polling mode).

• Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Driver not installed

Parameters

- `uart_num`: UART number

`esp_err_t uart_set_loop_back (uart_port_t uart_num, bool loop_back_en)`

Configure TX signal loop back to RX module, just for the test usage.

• Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Driver not installed

Parameters

- `uart_num`: UART number
- `loop_back_en`: Set true to enable the loop back function, else set it false.

void `uart_set_always_rx_timeout (uart_port_t uart_num, bool always_rx_timeout_en)`

Configure behavior of UART RX timeout interrupt.

When `always_rx_timeout` is true, timeout interrupt is triggered even if FIFO is full. This function can cause extra timeout interrupts triggered only to send the timeout event. Call this function only if you want to ensure timeout interrupt will always happen after a byte stream.

Parameters

- `uart_num`: UART number
- `always_rx_timeout_en`: Set to false enable the default behavior of timeout interrupt, set it to true to always trigger timeout interrupt.

Structures

struct `uart_intr_config_t`

UART interrupt configuration parameters for `uart_intr_config` function.

Public Members

`uint32_t intr_enable_mask`

UART interrupt enable mask, choose from `UART_XXXX_INT_ENA_M` under `UART_INT_ENA_REG(i)`, connect with bit-or operator

`uint8_t rx_timeout_thresh`

UART timeout interrupt threshold (unit: time of sending one byte)

`uint8_t txfifo_empty_intr_thresh`

UART TX empty interrupt threshold.

`uint8_t rxfifo_full_thresh`

UART RX full interrupt threshold.

struct `uart_event_t`

Event structure used in UART event queue.

Public Members

`uart_event_type_t type`

UART event type

`size_t size`

UART data size for `UART_DATA` event

`bool timeout_flag`

UART data read timeout flag for `UART_DATA` event (no new data received during configured RX TOUT) If the event is caused by FIFO-full interrupt, then there will be no event with the timeout flag before the next byte coming.

Macros

`UART_NUM_0`

UART port 0

`UART_NUM_1`

UART port 1

`UART_NUM_MAX`

UART port max

`UART_PIN_NO_CHANGE`

Constant for `uart_set_pin` function which indicates that UART pin should not be changed

Type Definitions

typedef `intr_handle_t` `uart_isr_handle_t`

Enumerations

enum `uart_event_type_t`

UART event types used in the ring buffer.

Values:

UART_DATA
UART data event

UART_BREAK
UART break event

UART_BUFFER_FULL
UART RX buffer full event

UART_FIFO_OVF
UART FIFO overflow event

UART_FRAME_ERR
UART RX frame error event

UART_PARITY_ERR
UART RX parity event

UART_DATA_BREAK
UART TX data and break event

UART_PATTERN_DET
UART pattern detected

UART_EVENT_MAX
UART event max index

Header File

- [soc/include/hal/uart_types.h](#)

Structures

struct uart_at_cmd_t

UART AT cmd char configuration parameters Note that this function may different on different chip. Please refer to the TRM at configuration.

Public Members

uint8_t cmd_char
UART AT cmd char

uint8_t char_num
AT cmd char repeat number

uint32_t gap_tout
gap time(in baud-rate) between AT cmd char

uint32_t pre_idle
the idle time(in baud-rate) between the non AT char and first AT char

uint32_t post_idle
the idle time(in baud-rate) between the last AT char and the none AT char

struct uart_sw_flowctrl_t

UART software flow control configuration parameters.

Public Members

uint8_t xon_char
Xon flow control char

uint8_t xoff_char
Xoff flow control char

`uint8_t xon_thrd`

If the software flow control is enabled and the data amount in rxfifo is less than `xon_thrd`, an `xon_char` will be sent

`uint8_t xoff_thrd`

If the software flow control is enabled and the data amount in rxfifo is more than `xoff_thrd`, an `xoff_char` will be sent

struct uart_config_t

UART configuration parameters for `uart_param_config` function.

Public Members

`int baud_rate`

UART baud rate

`uart_word_length_t data_bits`

UART byte size

`uart_parity_t parity`

UART parity mode

`uart_stop_bits_t stop_bits`

UART stop bits

`uart_hw_flowcontrol_t flow_ctrl`

UART HW flow control mode (cts/rts)

`uint8_t rx_flow_ctrl_thresh`

UART HW RTS threshold

`uart_sclk_t source_clk`

UART source clock selection

Type Definitions

typedef int uart_port_t

UART port number, can be `UART_NUM_0 ~ (UART_NUM_MAX - 1)`.

Enumerations

enum uart_mode_t

UART mode selection.

Values:

`UART_MODE_UART = 0x00`

mode: regular UART mode

`UART_MODE_RS485_HALF_DUPLEX = 0x01`

mode: half duplex RS485 UART mode control by RTS pin

`UART_MODE_IRDA = 0x02`

mode: IRDA UART mode

`UART_MODE_RS485_COLLISION_DETECT = 0x03`

mode: RS485 collision detection UART mode (used for test purposes)

`UART_MODE_RS485_APP_CTRL = 0x04`

mode: application control RS485 UART mode (used for test purposes)

enum uart_word_length_t

UART word length constants.

Values:

UART_DATA_5_BITS = 0x0
word length: 5bits

UART_DATA_6_BITS = 0x1
word length: 6bits

UART_DATA_7_BITS = 0x2
word length: 7bits

UART_DATA_8_BITS = 0x3
word length: 8bits

UART_DATA_BITS_MAX = 0x4

enum uart_stop_bits_t

UART stop bits number.

Values:

UART_STOP_BITS_1 = 0x1
stop bit: 1bit

UART_STOP_BITS_1_5 = 0x2
stop bit: 1.5bits

UART_STOP_BITS_2 = 0x3
stop bit: 2bits

UART_STOP_BITS_MAX = 0x4

enum uart_parity_t

UART parity constants.

Values:

UART_PARITY_DISABLE = 0x0
Disable UART parity

UART_PARITY_EVEN = 0x2
Enable UART even parity

UART_PARITY_ODD = 0x3
Enable UART odd parity

enum uart_hw_flowcontrol_t

UART hardware flow control modes.

Values:

UART_HW_FLOWCTRL_DISABLE = 0x0
disable hardware flow control

UART_HW_FLOWCTRL_RTS = 0x1
enable RX hardware flow control (rts)

UART_HW_FLOWCTRL_CTS = 0x2
enable TX hardware flow control (cts)

UART_HW_FLOWCTRL_CTS_RTS = 0x3
enable hardware flow control

UART_HW_FLOWCTRL_MAX = 0x4

enum uart_signal_inv_t

UART signal bit map.

Values:

UART_SIGNAL_IRDA_TX_INV = (0x1 << 0)
inverse the UART irda_tx signal

UART_SIGNAL_IRDA_RX_INV = (0x1 << 1)
inverse the UART irda_rx signal

UART_SIGNAL_RXD_INV = (0x1 << 2)
inverse the UART rxd signal

UART_SIGNAL_CTS_INV = (0x1 << 3)
inverse the UART cts signal

UART_SIGNAL_DSR_INV = (0x1 << 4)
inverse the UART dsr signal

UART_SIGNAL_TXD_INV = (0x1 << 5)
inverse the UART txd signal

UART_SIGNAL_RTS_INV = (0x1 << 6)
inverse the UART rts signal

UART_SIGNAL_DTR_INV = (0x1 << 7)
inverse the UART dtr signal

enum uart_sclk_t

UART source clock.

Values:

UART_SCLK_APB = 0x0
UART source clock from APB

UART_SCLK_REF_TICK = 0x01
UART source clock from REF_TICK

GPIO Lookup Macros The UART peripherals have dedicated IO_MUX pins to which they are connected directly. However, signals can also be routed to other pins using the less direct GPIO matrix. To use direct routes, you need to know which pin is a dedicated IO_MUX pin for a UART channel. GPIO Lookup Macros simplify the process of finding and assigning IO_MUX pins. You choose a macro based on either the IO_MUX pin number, or a required UART channel name, and the macro will return the matching counterpart for you. See some examples below.

Note: These macros are useful if you need very high UART baud rates (over 40 MHz), which means you will have to use IO_MUX pins only. In other cases, these macros can be ignored, and you can use the GPIO Matrix as it allows you to configure any GPIO pin for any UART function.

1. [UART_NUM_2_TXD_DIRECT_GPIO_NUM](#) returns the IO_MUX pin number of UART channel 2 TXD pin (pin 17)
2. [UART_GPIO19_DIRECT_CHANNEL](#) returns the UART number of GPIO 19 when connected to the UART peripheral via IO_MUX (this is UART_NUM_0)
3. [UART_CTS_GPIO19_DIRECT_CHANNEL](#) returns the UART number of GPIO 19 when used as the UART CTS pin via IO_MUX (this is UART_NUM_0). Similar to the above macro but specifies the pin function which is also part of the IO_MUX assignment.

Header File

- [soc/esp32/include/soc/uart_channel.h](#)

Macros

UART_GPIO1_DIRECT_CHANNEL

UART_NUM_0_TXD_DIRECT_GPIO_NUM

UART_GPIO3_DIRECT_CHANNEL

UART_NUM_0_RXD_DIRECT_GPIO_NUM

UART_GPIO19_DIRECT_CHANNEL
UART_NUM_0_CTS_DIRECT_GPIO_NUM
UART_GPIO22_DIRECT_CHANNEL
UART_NUM_0_RTS_DIRECT_GPIO_NUM
UART_TXD_GPIO1_DIRECT_CHANNEL
UART_RXD_GPIO3_DIRECT_CHANNEL
UART_CTS_GPIO19_DIRECT_CHANNEL
UART_RTS_GPIO22_DIRECT_CHANNEL
UART_GPIO10_DIRECT_CHANNEL
UART_NUM_1_TXD_DIRECT_GPIO_NUM
UART_GPIO9_DIRECT_CHANNEL
UART_NUM_1_RXD_DIRECT_GPIO_NUM
UART_GPIO6_DIRECT_CHANNEL
UART_NUM_1_CTS_DIRECT_GPIO_NUM
UART_GPIO11_DIRECT_CHANNEL
UART_NUM_1_RTS_DIRECT_GPIO_NUM
UART_TXD_GPIO10_DIRECT_CHANNEL
UART_RXD_GPIO9_DIRECT_CHANNEL
UART_CTS_GPIO6_DIRECT_CHANNEL
UART_RTS_GPIO11_DIRECT_CHANNEL
UART_GPIO17_DIRECT_CHANNEL
UART_NUM_2_TXD_DIRECT_GPIO_NUM
UART_GPIO16_DIRECT_CHANNEL
UART_NUM_2_RXD_DIRECT_GPIO_NUM
UART_GPIO8_DIRECT_CHANNEL
UART_NUM_2_CTS_DIRECT_GPIO_NUM
UART_GPIO7_DIRECT_CHANNEL
UART_NUM_2_RTS_DIRECT_GPIO_NUM
UART_TXD_GPIO17_DIRECT_CHANNEL
UART_RXD_GPIO16_DIRECT_CHANNEL
UART_CTS_GPIO8_DIRECT_CHANNEL
UART_RTS_GPIO7_DIRECT_CHANNEL

Code examples for this API section are provided in the [peripherals](#) directory of ESP-IDF examples.

2.4 Application Protocols

2.4.1 ASIO port

Overview

Asio is a cross-platform C++ library, see <https://think-async.com>. It provides a consistent asynchronous model using a modern C++ approach.

ASIO documentation Please refer to the original asio documentation at <https://think-async.com/Asio/Documentation>. Asio also comes with a number of examples which could be find under Documentation/Examples on that web site.

Supported features ESP platform port currently supports only network asynchronous socket operations; does not support serial port and ssl. Internal asio settings for ESP include

- EXCEPTIONS are enabled in ASIO if enabled in menuconfig
- TYPEID is enabled in ASIO if enabled in menuconfig

Application Example

ESP examples are based on standard asio [protocols/asio](#):

- [protocols/asio/udp_echo_server](#)
- [protocols/asio/tcp_echo_server](#)
- [protocols/asio/chat_client](#)
- [protocols/asio/chat_server](#)

Please refer to the specific example README.md for details

2.4.2 ESP-MQTT

Overview

ESP-MQTT is an implementation of MQTT protocol client (MQTT is a lightweight publish/subscribe messaging protocol).

Features

- Supports MQTT over TCP, SSL with mbedtls, MQTT over Websocket, MQTT over Websocket Secure.
- Easy to setup with URI
- Multiple instances (Multiple clients in one application)
- Support subscribing, publishing, authentication, last will messages, keep alive pings and all 3 QoS levels (it should be a fully functional client).

Application Example

- [protocols/mqtt/tcp](#): MQTT over tcp, default port 1883
- [protocols/mqtt/ssl](#): MQTT over tcp, default port 8883
- [protocols/mqtt/ssl_psk](#): MQTT over tcp using pre-shared keys for authentication, default port 8883
- [protocols/mqtt/ws](#): MQTT over Websocket, default port 80
- [protocols/mqtt/wss](#): MQTT over Websocket Secure, default port 443

Configuration

URI

- Currently support `mqtt`, `mqtt`s, `ws`, `wss` schemes
- MQTT over TCP samples:
 - `mqtt://mqtt.eclipse.org`: MQTT over TCP, default port 1883:
 - `mqtt://mqtt.eclipse.org:1884` MQTT over TCP, port 1884:
 - `mqtt://username:password@mqtt.eclipse.org:1884` MQTT over TCP, port 1884, with username and password
- MQTT over SSL samples:
 - `mqtt`s://mqtt.eclipse.org: MQTT over SSL, port 8883
 - `mqtt`s://mqtt.eclipse.org:8884: MQTT over SSL, port 8884
- MQTT over Websocket samples:
 - `ws://mqtt.eclipse.org:80/mqtt`
- MQTT over Websocket Secure samples:
 - `wss://mqtt.eclipse.org:443/mqtt`
- Minimal configurations:

```
const esp_mqtt_client_config_t mqtt_cfg = {
    .uri = "mqtt://mqtt.eclipse.org",
    // .user_context = (void *)your_context
};
esp_mqtt_client_handle_t client = esp_mqtt_client_init(&mqtt_cfg);
esp_mqtt_client_register_event(client, ESP_EVENT_ANY_ID, mqtt_event_handler,
↪client);
esp_mqtt_client_start(client);
```

- Note: By default mqtt client uses event loop library to post related mqtt events (connected, subscribed, published, etc.)

SSL

- Get certificate from server, example: `mqtt.eclipse.org openssl s_client -showcerts -connect mqtt.eclipse.org:8883 </dev/null 2>/dev/null|openssl x509 -outform PEM >mqtt_eclipse_org.pem`
- Check the sample application: `examples/mqtt_ssl`
- Configuration:

```
const esp_mqtt_client_config_t mqtt_cfg = {
    .uri = "mqtts://mqtt.eclipse.org:8883",
    .event_handle = mqtt_event_handler,
    .cert_pem = (const char *)mqtt_eclipse_org_pem_start,
};
```

If the certificate is not null-terminated then `cert_len` should also be set. Other SSL related configuration parameters are:

- `use_global_ca_store`: use the global certificate store to verify server certificate, see `esp-tls.h` for more information
- `client_cert_pem`: pointer to certificate data in PEM or DER format for SSL mutual authentication, default is NULL, not required if mutual authentication is not needed.
- `client_cert_len`: length of the buffer pointed to by `client_cert_pem`. May be 0 for null-terminated pem.
- `client_key_pem`: pointer to private key data in PEM or DER format for SSL mutual authentication, default is NULL, not required if mutual authentication is not needed.
- `client_key_len`: length of the buffer pointed to by `client_key_pem`. May be 0 for null-terminated pem.
- `psk_hint_key`: pointer to PSK struct defined in `esp-tls.h` to enable PSK authentication (as alternative to certificate verification). If not NULL and server/client certificates are NULL, PSK is enabled
- `alpn_protos`: NULL-terminated list of protocols to be used for ALPN.

Last Will and Testament MQTT allows for a last will and testament (LWT) message to notify other clients when a client ungracefully disconnects. This is configured by the following fields in the `esp_mqtt_client_config_t` struct.

- `lwt_topic`: pointer to the LWT message topic
- `lwt_msg`: pointer to the LWT message
- `lwt_msg_len`: length of the LWT message, required if `lwt_msg` is not null-terminated
- `lwt_qos`: quality of service for the LWT message
- `lwt_retain`: specifies the retain flag of the LWT message

Other Configuration Parameters

- `disable_clean_session`: determines the clean session flag for the connect message, defaults to a clean session
- `keepalive`: determines how many seconds the client will wait for a ping response before disconnecting, default is 120 seconds.
- `disable_auto_reconnect`: enable to stop the client from reconnecting to server after errors or disconnects
- `user_context`: custom context that will be passed to the event handler
- `task_prio`: MQTT task priority, defaults to 5
- `task_stack`: MQTT task stack size, defaults to 6144 bytes, setting this will override setting from menu-config
- `buffer_size`: size of MQTT send/receive buffer, default is 1024 bytes
- `username`: pointer to the username used for connecting to the broker
- `password`: pointer to the password used for connecting to the broker
- `client_id`: pointer to the client id, defaults to `ESP32_%CHIPID%` where `%CHIPID%` are the last 3 bytes of MAC address in hex format
- `host`: MQTT broker domain (ipv4 as string), setting the uri will override this
- `port`: MQTT broker port, specifying the port in the uri will override this
- `transport`: sets the transport protocol, setting the uri will override this
- `refresh_connection_after_ms`: refresh connection after this value (in milliseconds)
- `event_handle`: handle for MQTT events as a callback in legacy mode
- `event_loop_handle`: handle for MQTT event loop library

For more options on `esp_mqtt_client_config_t`, please refer to API reference below

Change settings in Project Configuration Menu The settings for MQTT can be found using `idf.py menu-config`, under Component config -> ESP-MQTT Configuration

The following settings are available:

- `CONFIG_MQTT_PROTOCOL_311`: Enables 3.1.1 version of MQTT protocol
- `CONFIG_MQTT_TRANSPORT_SSL`, `CONFIG_MQTT_TRANSPORT_WEBSOCKET`: Enables specific MQTT transport layer, such as SSL, WEBSOCKET, WEBSOCKET_SECURE
- `CONFIG_MQTT_CUSTOM_OUTBOX`: Disables default implementation of `mqtt_outbox`, so a specific implementation can be supplied

Events

The following events may be posted by the MQTT client:

- `MQTT_EVENT_BEFORE_CONNECT`: The client is initialized and about to start connecting to the broker.
- `MQTT_EVENT_CONNECTED`: The client has successfully established a connection to the broker. The client is now ready to send and receive data.
- `MQTT_EVENT_DISCONNECTED`: The client has aborted the connection due to being unable to read or write data, e.g. because the server is unavailable.
- `MQTT_EVENT_SUBSCRIBED`: The broker has acknowledged the client's subscribe request. The event data will contain the message ID of the subscribe message.

- `MQTT_EVENT_UNSUBSCRIBED`: The broker has acknowledged the client's unsubscribe request. The event data will contain the message ID of the unsubscribe message.
- `MQTT_EVENT_PUBLISHED`: The broker has acknowledged the client's publish message. This will only be posted for Quality of Service level 1 and 2, as level 0 does not use acknowledgements. The event data will contain the message ID of the publish message.
- `MQTT_EVENT_DATA`: The client has received a publish message. The event data contains: message ID, name of the topic it was published to, received data and its length. For data that exceeds the internal buffer multiple `MQTT_EVENT_DATA` will be posted and `current_data_offset` and `total_data_len` from event data updated to keep track of the fragmented message.
- `MQTT_EVENT_ERROR`: The client has encountered an error. `esp_mqtt_error_type_t` from `error_handle` in the event data can be used to further determine the type of the error. The type of error will determine which parts of the `error_handle` struct is filled.

API Reference

Header File

- `mqtt/esp-mqtt/include/mqtt_client.h`

Functions

`esp_mqtt_client_handle_t esp_mqtt_client_init (const esp_mqtt_client_config_t *config)`

Creates mqtt client handle based on the configuration.

Return `mqtt_client_handle` if successfully created, `NULL` on error

Parameters

- `config`: mqtt configuration structure

`esp_err_t esp_mqtt_client_set_uri (esp_mqtt_client_handle_t client, const char *uri)`

Sets mqtt connection URI. This API is usually used to overrides the URI configured in `esp_mqtt_client_init`.

Return `ESP_FAIL` if URI parse error, `ESP_OK` on success

Parameters

- `client`: mqtt client handle
- `uri`:

`esp_err_t esp_mqtt_client_start (esp_mqtt_client_handle_t client)`

Starts mqtt client with already created client handle.

Return `ESP_OK` on success `ESP_ERR_INVALID_ARG` on wrong initialization `ESP_FAIL` on other error

Parameters

- `client`: mqtt client handle

`esp_err_t esp_mqtt_client_reconnect (esp_mqtt_client_handle_t client)`

This api is typically used to force reconnection upon a specific event.

Return `ESP_OK` on success `ESP_FAIL` if client is in invalid state

Parameters

- `client`: mqtt client handle

`esp_err_t esp_mqtt_client_disconnect (esp_mqtt_client_handle_t client)`

This api is typically used to force disconnection from the broker.

Return `ESP_OK` on success

Parameters

- `client`: mqtt client handle

`esp_err_t esp_mqtt_client_stop (esp_mqtt_client_handle_t client)`

Stops mqtt client tasks.

- Notes:
- Cannot be called from the mqtt event handler

Return `ESP_OK` on success `ESP_FAIL` if client is in invalid state

Parameters

- `client`: mqtt client handle

int **esp_mqtt_client_subscribe** (*esp_mqtt_client_handle_t* client, const char *topic, int qos)
Subscribe the client to defined topic with defined qos.

Notes:

- Client must be connected to send subscribe message
- This API is could be executed from a user task or from a mqtt event callback i.e. internal mqtt task (API is protected by internal mutex, so it might block if a longer data receive operation is in progress.

Return message_id of the subscribe message on success -1 on failure

Parameters

- `client`: mqtt client handle
- `topic`:
- `qos`:

int **esp_mqtt_client_unsubscribe** (*esp_mqtt_client_handle_t* client, const char *topic)
Unsubscribe the client from defined topic.

Notes:

- Client must be connected to send unsubscribe message
- It is thread safe, please refer to `esp_mqtt_client_subscribe` for details

Return message_id of the subscribe message on success -1 on failure

Parameters

- `client`: mqtt client handle
- `topic`:

int **esp_mqtt_client_publish** (*esp_mqtt_client_handle_t* client, const char *topic, const char *data, int len, int qos, int retain)
Client to send a publish message to the broker.

Notes:

- This API might block for several seconds, either due to network timeout (10s) or if publishing payloads longer than internal buffer (due to message fragmentation)
- Client doesn't have to be connected for this API to work, enqueueing the messages with qos>1 (returning -1 for all the qos=0 messages if disconnected). If `MQTT_SKIP_PUBLISH_IF_DISCONNECTED` is enabled, this API will not attempt to publish when the client is not connected and will always return -1.
- It is thread safe, please refer to `esp_mqtt_client_subscribe` for details

Return message_id of the publish message (for QoS 0 message_id will always be zero) on success. -1 on failure.

Parameters

- `client`: mqtt client handle
- `topic`: topic string
- `data`: payload string (set to NULL, sending empty payload message)
- `len`: data length, if set to 0, length is calculated from payload string
- `qos`: qos of publish message
- `retain`: retain flag

int **esp_mqtt_client_enqueue** (*esp_mqtt_client_handle_t* client, const char *topic, const char *data, int len, int qos, int retain, bool store)

Enqueue a message to the outbox, to be sent later. Typically used for messages with qos>0, but could be also used for qos=0 messages if store=true.

This API generates and stores the publish message into the internal outbox and the actual sending to the network is performed in the mqtt-task context (in contrast to the `esp_mqtt_client_publish()` which sends the publish message immediately in the user task's context). Thus, it could be used as a non blocking version of `esp_mqtt_client_publish()`.

Return message_id if queued successfully, -1 otherwise

Parameters

- `client`: mqtt client handle

- `topic`: topic string
- `data`: payload string (set to NULL, sending empty payload message)
- `len`: data length, if set to 0, length is calculated from payload string
- `qos`: qos of publish message
- `retain`: retain flag
- `store`: if true, all messages are enqueued; otherwise only qos1 and qos 2 are enqueued

esp_err_t **esp_mqtt_client_destroy** (*esp_mqtt_client_handle_t* client)

Destroys the client handle.

Notes:

- Cannot be called from the mqtt event handler

Return ESP_OK

Parameters

- `client`: mqtt client handle

esp_err_t **esp_mqtt_set_config** (*esp_mqtt_client_handle_t* client, **const** *esp_mqtt_client_config_t* *config)

Set configuration structure, typically used when updating the config (i.e. on “before_connect” event.

Return ESP_ERR_NO_MEM if failed to allocate ESP_OK on success

Parameters

- `client`: mqtt client handle
- `config`: mqtt configuration structure

esp_err_t **esp_mqtt_client_register_event** (*esp_mqtt_client_handle_t* client, *esp_mqtt_event_id_t* event, *esp_event_handler_t* event_handler, void *event_handler_arg)

Registers mqtt event.

Return ESP_ERR_NO_MEM if failed to allocate ESP_OK on success

Parameters

- `client`: mqtt client handle
- `event`: event type
- `event_handler`: handler callback
- `event_handler_arg`: handlers context

int **esp_mqtt_client_get_outbox_size** (*esp_mqtt_client_handle_t* client)

Get outbox size.

Return outbox size

Parameters

- `client`: mqtt client handle

Structures

struct esp_mqtt_error_codes

MQTT error code structure to be passed as a contextual information into ERROR event.

Important: This structure extends *esp_tls_last_error* error structure and is backward compatible with it (so might be down-casted and treated as *esp_tls_last_error* error, but recommended to update applications if used this way previously)

Use this structure directly checking `error_type` first and then appropriate error code depending on the source of the error:

| error_type | related member variables | note |
|------------------------------------|--|--|
| MQTT_ERROR_TYPE_TCP_TRANSPORT | esp_tls_last_esp_err, esp_tls_stack_err, esp_tls_cert_verify_flags, sock_errno | Error reported from tcp_transport/esp-tls |
| MQTT_ERROR_TYPE_CONNECTION_REFUSED | connect_return_code | Internal error reported from MQTT broker on connection |

Public Members

esp_err_t **esp_tls_last_esp_err**

last esp_err code reported from esp-tls component

int **esp_tls_stack_err**

tls specific error code reported from underlying tls stack

int **esp_tls_cert_verify_flags**

tls flags reported from underlying tls stack during certificate verification

esp_mqtt_error_type_t **error_type**

error type referring to the source of the error

esp_mqtt_connect_return_code_t **connect_return_code**

connection refused error code reported from MQTT broker on connection

int **esp_transport_sock_errno**

errno from the underlying socket

struct esp_mqtt_event_t

MQTT event configuration structure

Public Members

esp_mqtt_event_id_t **event_id**

MQTT event type

esp_mqtt_client_handle_t **client**

MQTT client handle for this event

void ***user_context**

User context passed from MQTT client config

char ***data**

Data associated with this event

int **data_len**

Length of the data for this event

int **total_data_len**

Total length of the data (longer data are supplied with multiple events)

int **current_data_offset**

Actual offset for the data associated with this event

char ***topic**

Topic associated with this event

int **topic_len**

Length of the topic for this event associated with this event

int **msg_id**

MQTT messaged id of message

int **session_present**

MQTT session_present flag for connection event

esp_mqtt_error_codes_t ***error_handle**

esp-mqtt error handle including esp-tls errors as well as internal mqtt errors

struct esp_mqtt_client_config_t

MQTT client configuration structure

Public Members

mqtt_event_callback_t **event_handle**

handle for MQTT events as a callback in legacy mode

esp_event_loop_handle_t **event_loop_handle**

handle for MQTT event loop library

const char *host

MQTT server domain (ipv4 as string)

const char *uri

Complete MQTT broker URI

uint32_t port

MQTT server port

const char *client_id

default client id is ESP32_CHIPID% where CHIPID% are last 3 bytes of MAC address in hex format

const char *username

MQTT username

const char *password

MQTT password

const char *lwt_topic

LWT (Last Will and Testament) message topic (NULL by default)

const char *lwt_msg

LWT message (NULL by default)

int lwt_qos

LWT message qos

int lwt_retain

LWT retained message flag

int lwt_msg_len

LWT message length

int disable_clean_session

mqtt clean session, default clean_session is true

int keepalive

mqtt keepalive, default is 120 seconds

bool disable_auto_reconnect

this mqtt client will reconnect to server (when errors/disconnect). Set disable_auto_reconnect=true to disable

void *user_context

pass user context to this option, then can receive that context in event->user_context

int task_prio

MQTT task priority, default is 5, can be changed in make menuconfig

int task_stack

MQTT task stack size, default is 6144 bytes, can be changed in make menuconfig

int buffer_size

size of MQTT send/receive buffer, default is 1024 (only receive buffer size if out_buffer_size defined)

const char *cert_pem

Pointer to certificate data in PEM or DER format for server verify (with SSL), default is NULL, not required to verify the server. PEM-format must have a terminating NULL-character. DER-format requires the length to be passed in cert_len.

size_t cert_len

Length of the buffer pointed to by cert_pem. May be 0 for null-terminated pem

const char *client_cert_pem

Pointer to certificate data in PEM or DER format for SSL mutual authentication, default is NULL, not required if mutual authentication is not needed. If it is not NULL, also client_key_pem has to be provided. PEM-format must have a terminating NULL-character. DER-format requires the length to be passed in client_cert_len.

size_t client_cert_len

Length of the buffer pointed to by client_cert_pem. May be 0 for null-terminated pem

const char *client_key_pem

Pointer to private key data in PEM or DER format for SSL mutual authentication, default is NULL, not required if mutual authentication is not needed. If it is not NULL, also client_cert_pem has to be provided. PEM-format must have a terminating NULL-character. DER-format requires the length to be passed in client_key_len

size_t client_key_len

Length of the buffer pointed to by client_key_pem. May be 0 for null-terminated pem

***esp_mqtt_transport_t* transport**

overrides URI transport

int refresh_connection_after_ms

Refresh connection after this value (in milliseconds)

const struct *psk_key_hint* *psk_hint_key

Pointer to PSK struct defined in esp_tls.h to enable PSK authentication (as alternative to certificate verification). If not NULL and server/client certificates are NULL, PSK is enabled

bool use_global_ca_store

Use a global ca_store for all the connections in which this bool is set.

int reconnect_timeout_ms

Reconnect to the broker after this value in milliseconds if auto reconnect is not disabled (defaults to 10s)

const char **alpn_protos

NULL-terminated list of supported application protocols to be used for ALPN

const char *clientkey_password

Client key decryption password string

int clientkey_password_len

String length of the password pointed to by clientkey_password

***esp_mqtt_protocol_ver_t* protocol_ver**

MQTT protocol version used for connection, defaults to value from menuconfig

int out_buffer_size

size of MQTT output buffer. If not defined, both output and input buffers have the same size defined as buffer_size

bool skip_cert_common_name_check

Skip any validation of server certificate CN field, this reduces the security of TLS and makes the mqtt client susceptible to MITM attacks

bool use_secure_element

enable secure element for enabling SSL connection

void *ds_data

carrier of handle for digital signature parameters

int network_timeout_ms

Abort network operation if it is not completed after this value, in milliseconds (defaults to 10s)

bool disable_keepalive

Set disable_keepalive=true to turn off keep-alive mechanism, false by default (keepalive is active by

default). Note: setting the config value `keepalive` to 0 doesn't disable keepalive feature, but uses a default keepalive period

Macros

MQTT_ERROR_TYPE_ESP_TLS

`MQTT_ERROR_TYPE_TCP_TRANSPORT` error type hold all sorts of transport layer errors, including ESP-TLS error, but in the past only the errors from `MQTT_ERROR_TYPE_ESP_TLS` layer were reported, so the ESP-TLS error type is re-defined here for backward compatibility

Type Definitions

```
typedef struct esp_mqtt_client *esp_mqtt_client_handle_t
```

```
typedef struct esp_mqtt_error_codes esp_mqtt_error_codes_t
```

MQTT error code structure to be passed as a contextual information into ERROR event.

Important: This structure extends `esp_tls_last_error` error structure and is backward compatible with it (so might be down-casted and treated as `esp_tls_last_error` error, but recommended to update applications if used this way previously)

Use this structure directly checking `error_type` first and then appropriate error code depending on the source of the error:

| | | | |
|---|--|---|--|
| <code>error_type</code> | related member variables | note | <code>MQTT_ERROR_TYPE_TCP_TRANSPORT</code> |
| <code>esp_tls_last_esp_err</code> , <code>esp_tls_stack_err</code> , <code>esp_tls_cert_verify_flags</code> , <code>sock_errno</code> | Error reported from tcp_transport/esp-tls | <code>MQTT_ERROR_TYPE_CONNECTION_REFUSED</code> | <code>connect_return_code</code> |
| | Internal error reported from MQTT broker on connection | | |

```
typedef esp_mqtt_event_t *esp_mqtt_event_handle_t
```

```
typedef esp_err_t (*mqtt_event_callback_t) (esp_mqtt_event_handle_t event)
```

Enumerations

```
enum esp_mqtt_event_id_t
```

MQTT event types.

User event handler receives context data in `esp_mqtt_event_t` structure with

- `user_context` - user data from `esp_mqtt_client_config_t`
- `client` - mqtt client handle
- various other data depending on event type

Values:

```
MQTT_EVENT_ANY = -1
```

```
MQTT_EVENT_ERROR = 0
```

on error event, additional context: connection return code, error handle from `esp_tls` (if supported)

```
MQTT_EVENT_CONNECTED
```

connected event, additional context: `session_present` flag

```
MQTT_EVENT_DISCONNECTED
```

disconnected event

```
MQTT_EVENT_SUBSCRIBED
```

subscribed event, additional context: `msg_id`

```
MQTT_EVENT_UNSUBSCRIBED
```

unsubscribed event

```
MQTT_EVENT_PUBLISHED
```

published event, additional context: `msg_id`

```
MQTT_EVENT_DATA
```

data event, additional context:

- `msg_id` message id
- topic pointer to the received topic
- `topic_len` length of the topic
- data pointer to the received data
- `data_len` length of the data for this event
- `current_data_offset` offset of the current data for this event
- `total_data_len` total length of the data received Note: Multiple `MQTT_EVENT_DATA` could be fired for one message, if it is longer than internal buffer. In that case only first event contains topic pointer and length, other contain data only with current data length and current data offset updating.

MQTT_EVENT_BEFORE_CONNECT

The event occurs before connecting

MQTT_EVENT_DELETED

Notification on delete of one message from the internal outbox, if the message couldn't have been sent and acknowledged before expiring defined in `OUTBOX_EXPIRED_TIMEOUT_MS`. (events are not posted upon deletion of successfully acknowledged messages)

- This event id is posted only if `MQTT_REPORT_DELETED_MESSAGES==1`
- Additional context: `msg_id` (id of the deleted message).

enum esp_mqtt_connect_return_code_t

MQTT connection error codes propagated via ERROR event

Values:

MQTT_CONNECTION_ACCEPTED = 0

Connection accepted

MQTT_CONNECTION_REFUSE_PROTOCOL

MQTT connection refused reason: Wrong protocol

MQTT_CONNECTION_REFUSE_ID_REJECTED

MQTT connection refused reason: ID rejected

MQTT_CONNECTION_REFUSE_SERVER_UNAVAILABLE

MQTT connection refused reason: Server unavailable

MQTT_CONNECTION_REFUSE_BAD_USERNAME

MQTT connection refused reason: Wrong user

MQTT_CONNECTION_REFUSE_NOT_AUTHORIZED

MQTT connection refused reason: Wrong username or password

enum esp_mqtt_error_type_t

MQTT connection error codes propagated via ERROR event

Values:

MQTT_ERROR_TYPE_NONE = 0

MQTT_ERROR_TYPE_TCP_TRANSPORT

MQTT_ERROR_TYPE_CONNECTION_REFUSED

enum esp_mqtt_transport_t

Values:

MQTT_TRANSPORT_UNKNOWN = 0x0

MQTT_TRANSPORT_OVER_TCP

MQTT over TCP, using scheme: `mqtt`

MQTT_TRANSPORT_OVER_SSL

MQTT over SSL, using scheme: `mqttssl`

MQTT_TRANSPORT_OVER_WS

MQTT over Websocket, using scheme: `ws`

MQTT_TRANSPORT_OVER_WSSMQTT over Websocket Secure, using scheme: `wss`**enum esp_mqtt_protocol_ver_t**

MQTT protocol version used for connection

*Values:***MQTT_PROTOCOL_UNDEFINED** = 0**MQTT_PROTOCOL_V_3_1****MQTT_PROTOCOL_V_3_1_1**

2.4.3 ESP-TLS

Overview

The ESP-TLS component provides a simplified API interface for accessing the commonly used TLS functionality. It supports common scenarios like CA certification validation, SNI, ALPN negotiation, non-blocking connection among others. All the configuration can be specified in the `esp_tls_cfg_t` data structure. Once done, TLS communication can be conducted using the following APIs:

- `esp_tls_conn_new()`: for opening a new TLS connection.
- `esp_tls_conn_read()`: for reading from the connection.
- `esp_tls_conn_write()`: for writing into the connection.
- `esp_tls_conn_delete()`: for freeing up the connection.

Any application layer protocol like HTTP1, HTTP2 etc can be executed on top of this layer.

Application Example

Simple HTTPS example that uses ESP-TLS to establish a secure socket connection: [protocols/https_request](#).

Tree structure for ESP-TLS component

```

├── esp_tls.c
├── esp_tls.h
├── esp_tls_mbedtls.c
├── esp_tls_wolfssl.c
└── private_include
    ├── esp_tls_mbedtls.h
    └── esp_tls_wolfssl.h

```

The ESP-TLS component has a file `esp-tls/esp_tls.h` which contain the public API headers for the component. Internally ESP-TLS component uses one of the two SSL/TLS Libraries between `mbedtls` and `wolfssl` for its operation. API specific to `mbedtls` are present in `esp-tls/private_include/esp_tls_mbedtls.h` and API specific to `wolfssl` are present in `esp-tls/private_include/esp_tls_wolfssl.h`.

Underlying SSL/TLS Library Options

The ESP-TLS component has an option to use `mbedtls` or `wolfssl` as their underlying SSL/TLS library. By default only `mbedtls` is available and is used, `wolfssl` SSL/TLS library is available publicly at <https://github.com/espressif/esp-wolfssl>. The repository provides `wolfssl` component in binary format, it also provides few examples which are useful for understanding the API. Please refer the repository `README.md` for information on licensing and other options. Please see below option for using `wolfssl` in your project.

Note: *As the library options are internal to ESP-TLS, switching the libraries will not change ESP-TLS specific code for a project.*

How to use wolfssl with ESP-IDF

There are two ways to use wolfssl in your project

- 1) Directly add wolfssl as a component in your project with following three commands.:

```
(First change directory (cd) to your project directory)
mkdir components
cd components
git clone https://github.com/espressif/esp-wolfssl.git
```

- 2) Add wolfssl as an extra component in your project.

- Download wolfssl with:

```
git clone https://github.com/espressif/esp-wolfssl.git
```

- Include esp-wolfssl in ESP-IDF with setting EXTRA_COMPONENT_DIRS in CMakeLists.txt/Makefile of your project as done in [wolfssl/examples](#). For reference see Optional Project variables in [build-system](#).

After above steps, you will have option to choose wolfssl as underlying SSL/TLS library in configuration menu of your project as follows:

```
idf.py/make menuconfig -> ESP-TLS -> choose SSL/TLS Library -> mbedtls/wolfssl
```

Comparison between mbedtls and wolfssl

The following table shows a typical comparison between wolfssl and mbedtls when [protocols/https_request](#) example (which has server authentication) was run with both SSL/TLS libraries and with all respective configurations set to default. (mbedtls IN_CONTENT length and OUT_CONTENT length were set to 16384 bytes and 4096 bytes respectively)

| Property | Wolfssl | Mbedtls |
|---------------------|---------|---------|
| Total Heap Consumed | ~19 Kb | ~37 Kb |
| Task Stack Used | ~2.2 Kb | ~3.6 Kb |
| Bin size | ~858 Kb | ~736 Kb |

Note: *These values are subject to change with change in configuration options and version of respective libraries.*

API Reference

Header File

- [esp-tls/esp_tls.h](#)

Functions

`esp_tls_t *esp_tls_init` (void)

Create TLS connection.

This function allocates and initializes esp-tls structure handle.

Return tls Pointer to esp-tls as esp-tls handle if successfully initialized, NULL if allocation error

*esp_tls_t****esp_tls_conn_new**(**const** char *hostname, int hostlen, int port, **const** *esp_tls_cfg_t* *cfg)

Create a new blocking TLS/SSL connection.

This function establishes a TLS/SSL connection with the specified host in blocking manner.

Note: This API is present for backward compatibility reasons. Alternative function with the same functionality is *esp_tls_conn_new_sync* (and its asynchronous version *esp_tls_conn_new_async*)

Return pointer to *esp_tls_t*, or NULL if connection couldn't be opened.

Parameters

- [in] hostname: Hostname of the host.
- [in] hostlen: Length of hostname.
- [in] port: Port number of the host.
- [in] cfg: TLS configuration as *esp_tls_cfg_t*. If you wish to open non-TLS connection, keep this NULL. For TLS connection, a pass pointer to *esp_tls_cfg_t*. At a minimum, this structure should be zero-initialized.

int **esp_tls_conn_new_sync**(**const** char *hostname, int hostlen, int port, **const** *esp_tls_cfg_t* *cfg, *esp_tls_t* *tls)

Create a new blocking TLS/SSL connection.

This function establishes a TLS/SSL connection with the specified host in blocking manner.

Return

- -1 If connection establishment fails.
- 1 If connection establishment is successful.
- 0 If connection state is in progress.

Parameters

- [in] hostname: Hostname of the host.
- [in] hostlen: Length of hostname.
- [in] port: Port number of the host.
- [in] cfg: TLS configuration as *esp_tls_cfg_t*. If you wish to open non-TLS connection, keep this NULL. For TLS connection, a pass pointer to *esp_tls_cfg_t*. At a minimum, this structure should be zero-initialized.
- [in] tls: Pointer to *esp_tls* as *esp_tls* handle.

*esp_tls_t****esp_tls_conn_http_new**(**const** char *url, **const** *esp_tls_cfg_t* *cfg)

Create a new blocking TLS/SSL connection with a given "HTTP" url.

The behaviour is same as *esp_tls_conn_new*() API. However this API accepts host's url.

Return pointer to *esp_tls_t*, or NULL if connection couldn't be opened.

Parameters

- [in] url: url of host.
- [in] cfg: TLS configuration as *esp_tls_cfg_t*. If you wish to open non-TLS connection, keep this NULL. For TLS connection, a pass pointer to '*esp_tls_cfg_t*'. At a minimum, this structure should be zero-initialized.

int **esp_tls_conn_new_async**(**const** char *hostname, int hostlen, int port, **const** *esp_tls_cfg_t* *cfg, *esp_tls_t* *tls)

Create a new non-blocking TLS/SSL connection.

This function initiates a non-blocking TLS/SSL connection with the specified host, but due to its non-blocking nature, it doesn't wait for the connection to get established.

Return

- -1 If connection establishment fails.
- 0 If connection establishment is in progress.
- 1 If connection establishment is successful.

Parameters

- [in] hostname: Hostname of the host.
- [in] hostlen: Length of hostname.
- [in] port: Port number of the host.

- [in] `cfg`: TLS configuration as `esp_tls_cfg_t`. `non_block` member of this structure should be set to be true.
- [in] `tls`: pointer to esp-tls as esp-tls handle.

int **esp_tls_conn_http_new_async** (const char *url, const *esp_tls_cfg_t* *cfg, *esp_tls_t* *tls)
Create a new non-blocking TLS/SSL connection with a given “HTTP” url.

The behaviour is same as `esp_tls_conn_new()` API. However this API accepts host’s url.

Return

- -1 If connection establishment fails.
- 0 If connection establishment is in progress.
- 1 If connection establishment is successful.

Parameters

- [in] `url`: url of host.
- [in] `cfg`: TLS configuration as `esp_tls_cfg_t`.
- [in] `tls`: pointer to esp-tls as esp-tls handle.

static ssize_t **esp_tls_conn_write** (*esp_tls_t* *tls, const void *data, size_t datalen)
Write from buffer ‘data’ into specified tls connection.

Return

- >0 if write operation was successful, the return value is the number of bytes actually written to the TLS/SSL connection.
- 0 if write operation was not successful. The underlying connection was closed.
- <0 if write operation was not successful, because either an error occurred or an action must be taken by the calling process.
- `ESP_TLS_ERR_SSL_WANT_READ/ESP_TLS_ERR_SSL_WANT_WRITE`. if the handshake is incomplete and waiting for data to be available for reading. In this case this functions needs to be called again when the underlying transport is ready for operation.

Parameters

- [in] `tls`: pointer to esp-tls as esp-tls handle.
- [in] `data`: Buffer from which data will be written.
- [in] `datalen`: Length of data buffer.

static ssize_t **esp_tls_conn_read** (*esp_tls_t* *tls, void *data, size_t datalen)
Read from specified tls connection into the buffer ‘data’ .

Return

- >0 if read operation was successful, the return value is the number of bytes actually read from the TLS/SSL connection.
- 0 if read operation was not successful. The underlying connection was closed.
- <0 if read operation was not successful, because either an error occurred or an action must be taken by the calling process.

Parameters

- [in] `tls`: pointer to esp-tls as esp-tls handle.
- [in] `data`: Buffer to hold read data.
- [in] `datalen`: Length of data buffer.

void **esp_tls_conn_delete** (*esp_tls_t* *tls)
Close the TLS/SSL connection and free any allocated resources.

This function should be called to close each tls connection opened with `esp_tls_conn_new()` or `esp_tls_conn_http_new()` APIs.

Parameters

- [in] `tls`: pointer to esp-tls as esp-tls handle.

ssize_t **esp_tls_get_bytes_avail** (*esp_tls_t* *tls)
Return the number of application data bytes remaining to be read from the current record.

This API is a wrapper over mbedtls’s `mbedtls_ssl_get_bytes_avail()` API.

Return

- -1 in case of invalid arg

- bytes available in the application data record read buffer

Parameters

- [in] `tls`: pointer to esp-tls as esp-tls handle.

esp_err_t **esp_tls_init_global_ca_store** (void)

Create a global CA store, initially empty.

This function should be called if the application wants to use the same CA store for multiple connections. This function initialises the global CA store which can be then set by calling `esp_tls_set_global_ca_store()`. To be effective, this function must be called before any call to `esp_tls_set_global_ca_store()`.

Return

- ESP_OK if creating global CA store was successful.
- ESP_ERR_NO_MEM if an error occurred when allocating the mbedTLS resources.

esp_err_t **esp_tls_set_global_ca_store** (const unsigned char **cacert_pem_buf*, const unsigned int *cacert_pem_bytes*)

Set the global CA store with the buffer provided in pem format.

This function should be called if the application wants to set the global CA store for multiple connections i.e. to add the certificates in the provided buffer to the certificate chain. This function implicitly calls `esp_tls_init_global_ca_store()` if it has not already been called. The application must call this function before calling `esp_tls_conn_new()`.

Return

- ESP_OK if adding certificates was successful.
- Other if an error occurred or an action must be taken by the calling process.

Parameters

- [in] `cacert_pem_buf`: Buffer which has certificates in pem format. This buffer is used for creating a global CA store, which can be used by other tls connections.
- [in] `cacert_pem_bytes`: Length of the buffer.

void **esp_tls_free_global_ca_store** (void)

Free the global CA store currently being used.

The memory being used by the global CA store to store all the parsed certificates is freed up. The application can call this API if it no longer needs the global CA store.

esp_err_t **esp_tls_get_and_clear_last_error** (*esp_tls_error_handle_t* *h*, int **esp_tls_code*, int **esp_tls_flags*)

Returns last error in *esp_tls* with detailed mbedtls related error codes. The error information is cleared internally upon return.

Return

- ESP_ERR_INVALID_STATE if invalid parameters
- ESP_OK (0) if no error occurred
- specific error code (based on ESP_ERR_ESP_TLS_BASE) otherwise

Parameters

- [in] `h`: esp-tls error handle.
- [out] `esp_tls_code`: last error code returned from mbedtls api (set to zero if none) This pointer could be NULL if caller does not care about `esp_tls_code`
- [out] `esp_tls_flags`: last certification verification flags (set to zero if none) This pointer could be NULL if caller does not care about `esp_tls_code`

Structures

struct esp_tls_last_error

Error structure containing relevant errors in case tls error occurred.

Public Members

esp_err_t **last_error**

error code (based on ESP_ERR_ESP_TLS_BASE) of the last occurred error

int esp_tls_error_code
esp_tls error code from last *esp_tls* failed api

int esp_tls_flags
last certification verification flags

struct psk_key_hint
ESP-TLS preshared key and hint structure.

Public Members

const uint8_t *key
key in PSK authentication mode in binary format

const size_t key_size
length of the key

const char *hint
hint in PSK authentication mode in string format

struct tls_keep_alive_cfg
Keep alive parameters structure.

Public Members

bool keep_alive_enable
Enable keep-alive timeout

int keep_alive_idle
Keep-alive idle time (second)

int keep_alive_interval
Keep-alive interval time (second)

int keep_alive_count
Keep-alive packet retry send count

struct esp_tls_cfg
ESP-TLS configuration parameters.

Note Note about format of certificates:

- This structure includes certificates of a Certificate Authority, of client or server as well as private keys, which may be of PEM or DER format. In case of PEM format, the buffer must be NULL terminated (with NULL character included in certificate size).
- Certificate Authority's certificate may be a chain of certificates in case of PEM format, but could be only one certificate in case of DER format
- Variables names of certificates and private key buffers and sizes are defined as unions providing backward compatibility for legacy *_pem_buf and *_pem_bytes names which suggested only PEM format was supported. It is encouraged to use generic names such as cacert_buf and cacert_bytes.

Public Members

const char **alpn_protos
Application protocols required for HTTP2. If HTTP2/ALPN support is required, a list of protocols that should be negotiated. The format is length followed by protocol name. For the most common cases the following is ok: `const char **alpn_protos = { "h2", NULL };`

- where 'h2' is the protocol name

const unsigned char *cacert_buf
Certificate Authority's certificate in a buffer. Format may be PEM or DER, depending on mbedtls-support This buffer should be NULL terminated in case of PEM

const unsigned char ***cacert_pem_buf**

CA certificate buffer legacy name

unsigned int **cacert_bytes**

Size of Certificate Authority certificate pointed to by cacert_buf (including NULL-terminator in case of PEM format)

unsigned int **cacert_pem_bytes**

Size of Certificate Authority certificate legacy name

const unsigned char ***clientcert_buf**

Client certificate in a buffer Format may be PEM or DER, depending on mbedtls-support This buffer should be NULL terminated in case of PEM

const unsigned char ***clientcert_pem_buf**

Client certificate legacy name

unsigned int **clientcert_bytes**

Size of client certificate pointed to by clientcert_pem_buf (including NULL-terminator in case of PEM format)

unsigned int **clientcert_pem_bytes**

Size of client certificate legacy name

const unsigned char ***clientkey_buf**

Client key in a buffer Format may be PEM or DER, depending on mbedtls-support This buffer should be NULL terminated in case of PEM

const unsigned char ***clientkey_pem_buf**

Client key legacy name

unsigned int **clientkey_bytes**

Size of client key pointed to by clientkey_pem_buf (including NULL-terminator in case of PEM format)

unsigned int **clientkey_pem_bytes**

Size of client key legacy name

const unsigned char ***clientkey_password**

Client key decryption password string

unsigned int **clientkey_password_len**

String length of the password pointed to by clientkey_password

bool **non_block**

Configure non-blocking mode. If set to true the underneath socket will be configured in non blocking mode after tls session is established

int **timeout_ms**

Network timeout in milliseconds

bool **use_global_ca_store**

Use a global ca_store for all the connections in which this bool is set.

const char ***common_name**

If non-NULL, server certificate CN must match this name. If NULL, server certificate CN must match hostname.

bool **skip_common_name**

Skip any validation of server certificate CN field

const *psk_hint_key_t* ***psk_hint_key**

Pointer to PSK hint and key. if not NULL (and certificates are NULL) then PSK authentication is enabled with configured setup. Important note: the pointer must be valid for connection

tls_keep_alive_cfg_t ***keep_alive_cfg**

Enable TCP keep-alive timeout for SSL connection

struct esp_tls

ESP-TLS Connection Handle.

Public Membersint **sockfd**

Underlying socket file descriptor.

ssize_t (***read**) (**struct esp_tls** *tls, char *data, size_t datalen)

Callback function for reading data from TLS/SSL connection.

ssize_t (***write**) (**struct esp_tls** *tls, const char *data, size_t datalen)

Callback function for writing data to TLS/SSL connection.

esp_tls_conn_state_t **conn_state**

ESP-TLS Connection state

fd_set **rset**

read file descriptors

fd_set **wset**

write file descriptors

bool **is_tls**

indicates connection type (TLS or NON-TLS)

esp_tls_role_t **role**

esp-tls role

- ESP_TLS_CLIENT
- ESP_TLS_SERVER

esp_tls_error_handle_t **error_handle**

handle to error descriptor

Macros**ESP_ERR_ESP_TLS_BASE**

Starting number of ESP-TLS error codes

ESP_ERR_ESP_TLS_CANNOT_RESOLVE_HOSTNAME

Error if hostname couldn't be resolved upon tls connection

ESP_ERR_ESP_TLS_CANNOT_CREATE_SOCKET

Failed to create socket

ESP_ERR_ESP_TLS_UNSUPPORTED_PROTOCOL_FAMILY

Unsupported protocol family

ESP_ERR_ESP_TLS_FAILED_CONNECT_TO_HOST

Failed to connect to host

ESP_ERR_ESP_TLS_SOCKET_SETOPT_FAILED

failed to set socket option

ESP_ERR_MBEDTLS_CERT_PARTLY_OK

mbedtls parse certificates was partly successful

ESP_ERR_MBEDTLS_CTR_DRBG_SEED_FAILED

mbedtls api returned error

ESP_ERR_MBEDTLS_SSL_SET_HOSTNAME_FAILED

mbedtls api returned error

ESP_ERR_MBEDTLS_SSL_CONFIG_DEFAULTS_FAILED

mbedtls api returned error

ESP_ERR_MBEDTLS_SSL_CONF_ALPN_PROTOCOLS_FAILED

mbedtls api returned error

ESP_ERR_MBEDTLS_X509_CRT_PARSE_FAILED

mbedtls api returned error

ESP_ERR_MBEDTLS_SSL_CONF_OWN_CERT_FAILED

mbedtls api returned error

ESP_ERR_MBEDTLS_SSL_SETUP_FAILED

mbedtls api returned error

ESP_ERR_MBEDTLS_SSL_WRITE_FAILED

mbedtls api returned error

ESP_ERR_MBEDTLS_PK_PARSE_KEY_FAILED

mbedtls api returned failed

ESP_ERR_MBEDTLS_SSL_HANDSHAKE_FAILED

mbedtls api returned failed

ESP_ERR_MBEDTLS_SSL_CONF_PSK_FAILED

mbedtls api returned failed

ESP_ERR_ESP_TLS_CONNECTION_TIMEOUT

new connection in esp_tls_low_level_conn connection timeouted

Type Definitions**typedef struct esp_tls_last_error *esp_tls_error_handle_t****typedef struct esp_tls_last_error esp_tls_last_error_t**

Error structure containing relevant errors in case tls error occurred.

typedef enum esp_tls_conn_state esp_tls_conn_state_t

ESP-TLS Connection State.

typedef enum esp_tls_role esp_tls_role_t**typedef struct psk_key_hint psk_hint_key_t**

ESP-TLS preshared key and hint structure.

typedef struct tls_keep_alive_cfg tls_keep_alive_cfg_t

Keep alive parameters structure.

typedef struct esp_tls_cfg esp_tls_cfg_t

ESP-TLS configuration parameters.

Note Note about format of certificates:

- This structure includes certificates of a Certificate Authority, of client or server as well as private keys, which may be of PEM or DER format. In case of PEM format, the buffer must be NULL terminated (with NULL character included in certificate size).
- Certificate Authority's certificate may be a chain of certificates in case of PEM format, but could be only one certificate in case of DER format
- Variables names of certificates and private key buffers and sizes are defined as unions providing backward compatibility for legacy *_pem_buf and *_pem_bytes names which suggested only PEM format was supported. It is encouraged to use generic names such as cacert_buf and cacert_bytes.

typedef struct esp_tls esp_tls_t

ESP-TLS Connection Handle.

Enumerations**enum esp_tls_conn_state**

ESP-TLS Connection State.

Values:


```

ESP_TLS_INIT = 0
ESP_TLS_CONNECTING
ESP_TLS_HANDSHAKE
ESP_TLS_FAIL
ESP_TLS_DONE
enum esp_tls_role
  Values:
ESP_TLS_CLIENT = 0
ESP_TLS_SERVER

```

2.4.4 ESP HTTP Client

Overview

`esp_http_client` provides an API for making HTTP/S requests from ESP-IDF programs. The steps to use this API for an HTTP request are:

- `esp_http_client_init()`: To use the HTTP client, the first thing we must do is create an `esp_http_client` by pass into this function with the `esp_http_client_config_t` configurations. Which configuration values we do not define, the library will use default.
- `esp_http_client_perform()`: The `esp_http_client` argument created from the init function is needed. This function performs all operations of the `esp_http_client`, from opening the connection, sending data, downloading data and closing the connection if necessary. All related events will be invoked in the event_handle (defined by `esp_http_client_config_t`). This function performs its job and blocks the current task until it's done
- `esp_http_client_cleanup()`: After completing our `esp_http_client`'s task, this is the last function to be called. It will close the connection (if any) and free up all the memory allocated to the HTTP client

Application Example

```

esp_err_t _http_event_handle(esp_http_client_event_t *evt)
{
    switch(evt->event_id) {
        case HTTP_EVENT_ERROR:
            ESP_LOGI(TAG, "HTTP_EVENT_ERROR");
            break;
        case HTTP_EVENT_ON_CONNECTED:
            ESP_LOGI(TAG, "HTTP_EVENT_ON_CONNECTED");
            break;
        case HTTP_EVENT_HEADER_SENT:
            ESP_LOGI(TAG, "HTTP_EVENT_HEADER_SENT");
            break;
        case HTTP_EVENT_ON_HEADER:
            ESP_LOGI(TAG, "HTTP_EVENT_ON_HEADER");
            printf("%.s", evt->data_len, (char*)evt->data);
            break;
        case HTTP_EVENT_ON_DATA:
            ESP_LOGI(TAG, "HTTP_EVENT_ON_DATA, len=%d", evt->data_len);
            if (!esp_http_client_is_chunked_response(evt->client)) {
                printf("%.s", evt->data_len, (char*)evt->data);
            }

            break;
        case HTTP_EVENT_ON_FINISH:
            ESP_LOGI(TAG, "HTTP_EVENT_ON_FINISH");

```

(continues on next page)

(continued from previous page)

```

        break;
    case HTTP_EVENT_DISCONNECTED:
        ESP_LOGI(TAG, "HTTP_EVENT_DISCONNECTED");
        break;
    }
    return ESP_OK;
}

esp_http_client_config_t config = {
    .url = "http://httpbin.org/redirect/2",
    .event_handler = _http_event_handle,
};
esp_http_client_handle_t client = esp_http_client_init(&config);
esp_err_t err = esp_http_client_perform(client);

if (err == ESP_OK) {
    ESP_LOGI(TAG, "Status = %d, content_length = %d",
        esp_http_client_get_status_code(client),
        esp_http_client_get_content_length(client));
}
esp_http_client_cleanup(client);

```

Persistent Connections

Persistent connections means that the HTTP client can re-use the same connection for several transfers. If the server does not request to close the connection with the `Connection: close` header, the new transfer with sample ip address, port, and protocol.

To allow the HTTP client to take full advantage of persistent connections, you should do as many of your file transfers as possible using the same handle.

Persistent Connections example

```

esp_err_t err;
esp_http_client_config_t config = {
    .url = "http://httpbin.org/get",
};
esp_http_client_handle_t client = esp_http_client_init(&config);
// first request
err = esp_http_client_perform(client);

// second request
esp_http_client_set_url(client, "http://httpbin.org/anything")
esp_http_client_set_method(client, HTTP_METHOD_DELETE);
esp_http_client_set_header(client, "HeaderKey", "HeaderValue");
err = esp_http_client_perform(client);

esp_http_client_cleanup(client);

```

HTTPS

The HTTP client supports SSL connections using `mbedtls`, with the `url` configuration starting with `https` scheme (or `transport_type = HTTP_TRANSPORT_OVER_SSL`). HTTPS support can be configured via `ref:CONFIG_ENABLE_HTTPS` (enabled by default)..

Note: By providing information using HTTPS, the library will use the SSL transport type to connect to the server. If you want to verify server, then need to provide additional certificate in PEM format, and provide to `cert_pem`

in `esp_http_client_config_t`

HTTPS example

```
static void https()
{
    esp_http_client_config_t config = {
        .url = "https://www.howsmyssl.com",
        .cert_pem = howsmyssl_com_root_cert_pem_start,
    };
    esp_http_client_handle_t client = esp_http_client_init(&config);
    esp_err_t err = esp_http_client_perform(client);

    if (err == ESP_OK) {
        ESP_LOGI(TAG, "Status = %d, content_length = %d",
            esp_http_client_get_status_code(client),
            esp_http_client_get_content_length(client));
    }
    esp_http_client_cleanup(client);
}
```

HTTP Stream

Some applications need to open the connection and control the reading of the data in an active manner. the HTTP client supports some functions to make this easier, of course, once you use these functions you should not use the `esp_http_client_perform()` function with that handle, and `esp_http_client_init()` always to be called first to get the handle. Perform that functions in the order below:

- `esp_http_client_init()`: to create and handle
- `esp_http_client_set_*` or `esp_http_client_delete_*`: to modify the http connection information (optional)
- `esp_http_client_open()`: Open the http connection with `write_len` parameter, `write_len=0` if we only need read
- `esp_http_client_write()`: Upload data, max length equal to `write_len` of `esp_http_client_open()` function. We may not need to call it if `write_len=0`
- `esp_http_client_fetch_headers()`: After sending the headers and write data (if any) to the server, this function will read the HTTP Server response headers. Calling this function will return the `content-length` from the Server, and we can call `esp_http_client_get_status_code()` for the HTTP status of the connection.
- `esp_http_client_read()`: Now, we can read the HTTP stream by this function.
- `esp_http_client_close()`: We should the connection after finish
- `esp_http_client_cleanup()`: And release the resources

Perform HTTP request as Stream reader Check the example function `http_perform_as_stream_reader` at [protocols/esp_http_client](#).

HTTP Authentication

The HTTP client supports both **Basic** and **Digest** Authentication. By providing usernames and passwords in `url` or in the `username`, `password` of `config` entry. And with `auth_type = HTTP_AUTH_TYPE_BASIC`, the HTTP client takes only 1 perform to pass the authentication process. If `auth_type = HTTP_AUTH_TYPE_NONE`, but there are `username` and `password` in the configuration, the HTTP client takes 2 performs. The first time it connects to the server and receives the UNAUTHORIZED header. Based on this information, it will know which authentication method to choose, and perform it on the second.

Config authentication example with URI

```
esp_http_client_config_t config = {
    .url = "http://user:passwd@httpbin.org/basic-auth/user/passwd",
    .auth_type = HTTP_AUTH_TYPE_BASIC,
};
```

Config authentication example with username, password entry

```
esp_http_client_config_t config = {
    .url = "http://httpbin.org/basic-auth/user/passwd",
    .username = "user",
    .password = "passwd",
    .auth_type = HTTP_AUTH_TYPE_BASIC,
};
```

HTTP Client example: [protocols/esp_http_client](#).

API Reference

Header File

- [esp_http_client/include/esp_http_client.h](#)

Functions

esp_http_client_handle_t **esp_http_client_init** (**const** *esp_http_client_config_t* *config)

Start a HTTP session This function must be the first function to call, and it returns a *esp_http_client_handle_t* that you must use as input to other functions in the interface. This call MUST have a corresponding call to *esp_http_client_cleanup* when the operation is complete.

Return

- *esp_http_client_handle_t*
- NULL if any errors

Parameters

- [in] config: The configurations, see *http_client_config_t*

esp_err_t **esp_http_client_perform** (*esp_http_client_handle_t* client)

Invoke this function after *esp_http_client_init* and all the options calls are made, and will perform the transfer as described in the options. It must be called with the same *esp_http_client_handle_t* as input as the *esp_http_client_init* call returned. *esp_http_client_perform* performs the entire request in either blocking or non-blocking manner. By default, the API performs request in a blocking manner and returns when done, or if it failed, and in non-blocking manner, it returns if EAGAIN/EWOULDBLOCK or EINPROGRESS is encountered, or if it failed. And in case of non-blocking request, the user may call this API multiple times unless request & response is complete or there is a failure. To enable non-blocking *esp_http_client_perform()*, *is_async* member of *esp_http_client_config_t* must be set while making a call to *esp_http_client_init()* API. You can do any amount of calls to *esp_http_client_perform* while using the same *esp_http_client_handle_t*. The underlying connection may be kept open if the server allows it. If you intend to transfer more than one file, you are even encouraged to do so. *esp_http_client* will then attempt to re-use the same connection for the following transfers, thus making the operations faster, less CPU intense and using less network resources. Just note that you will have to use *esp_http_client_set_** between the invokes to set options for the following *esp_http_client_perform*.

Note You must never call this function simultaneously from two places using the same client handle.

Let the function return first before invoking it another time. If you want parallel transfers, you must use several *esp_http_client_handle_t*. This function include *esp_http_client_open* -> *esp_http_client_write* -> *esp_http_client_fetch_headers* -> *esp_http_client_read* (and option) *esp_http_client_close*.

Return

- ESP_OK on successful
- ESP_FAIL on error

Parameters

- `client`: The `esp_http_client` handle

esp_err_t `esp_http_client_set_url` (*esp_http_client_handle_t* `client`, **const** char *`url`)

Set URL for client, when performing this behavior, the options in the URL will replace the old ones.

Return

- `ESP_OK`
- `ESP_FAIL`

Parameters

- [in] `client`: The `esp_http_client` handle
- [in] `url`: The url

esp_err_t `esp_http_client_set_post_field` (*esp_http_client_handle_t* `client`, **const** char *`data`,
int `len`)

Set post data, this function must be called before `esp_http_client_perform`. Note: The data parameter passed to this function is a pointer and this function will not copy the data.

Return

- `ESP_OK`
- `ESP_FAIL`

Parameters

- [in] `client`: The `esp_http_client` handle
- [in] `data`: post data pointer
- [in] `len`: post length

int `esp_http_client_get_post_field` (*esp_http_client_handle_t* `client`, char **`data`)

Get current post field information.

Return Size of post data

Parameters

- [in] `client`: The client
- [out] `data`: Point to post data pointer

esp_err_t `esp_http_client_set_header` (*esp_http_client_handle_t* `client`, **const** char *`key`, **const**
char *`value`)

Set http request header, this function must be called after `esp_http_client_init` and before any perform function.

Return

- `ESP_OK`
- `ESP_FAIL`

Parameters

- [in] `client`: The `esp_http_client` handle
- [in] `key`: The header key
- [in] `value`: The header value

esp_err_t `esp_http_client_get_header` (*esp_http_client_handle_t* `client`, **const** char *`key`, char
**`value`)

Get http request header. The value parameter will be set to NULL if there is no header which is same as the key specified, otherwise the address of header value will be assigned to value parameter. This function must be called after `esp_http_client_init`.

Return

- `ESP_OK`
- `ESP_FAIL`

Parameters

- [in] `client`: The `esp_http_client` handle
- [in] `key`: The header key
- [out] `value`: The header value

esp_err_t `esp_http_client_get_username` (*esp_http_client_handle_t* `client`, char **`value`)

Get http request username. The address of username buffer will be assigned to value parameter. This function must be called after `esp_http_client_init`.

Return

- ESP_OK
- ESP_ERR_INVALID_ARG

Parameters

- [in] `client`: The `esp_http_client` handle
- [out] `value`: The username value

`esp_err_t esp_http_client_set_username(esp_http_client_handle_t client, const char *username)`

Set http request username. The value of username parameter will be assigned to username buffer. If the username parameter is NULL then username buffer will be freed.

Return

- ESP_OK
- ESP_ERR_INVALID_ARG

Parameters

- [in] `client`: The `esp_http_client` handle
- [in] `username`: The username value

`esp_err_t esp_http_client_get_password(esp_http_client_handle_t client, char **value)`

Get http request password. The address of password buffer will be assigned to value parameter. This function must be called after `esp_http_client_init`.

Return

- ESP_OK
- ESP_ERR_INVALID_ARG

Parameters

- [in] `client`: The `esp_http_client` handle
- [out] `value`: The password value

`esp_err_t esp_http_client_set_password(esp_http_client_handle_t client, char *password)`

Set http request password. The value of password parameter will be assigned to password buffer. If the password parameter is NULL then password buffer will be freed.

Return

- ESP_OK
- ESP_ERR_INVALID_ARG

Parameters

- [in] `client`: The `esp_http_client` handle
- [in] `password`: The password value

`esp_err_t esp_http_client_set_auth_type(esp_http_client_handle_t client, esp_http_client_auth_type_t auth_type)`

Set http request auth_type.

Return

- ESP_OK
- ESP_ERR_INVALID_ARG

Parameters

- [in] `client`: The `esp_http_client` handle
- [in] `auth_type`: The `esp_http_client` auth type

`esp_err_t esp_http_client_set_method(esp_http_client_handle_t client, esp_http_client_method_t method)`

Set http request method.

Return

- ESP_OK
- ESP_ERR_INVALID_ARG

Parameters

- [in] `client`: The `esp_http_client` handle
- [in] `method`: The method

`esp_err_t esp_http_client_delete_header(esp_http_client_handle_t client, const char *key)`

Delete http request header.

Return

- ESP_OK
- ESP_FAIL

Parameters

- [in] `client`: The `esp_http_client` handle
- [in] `key`: The key

esp_err_t **esp_http_client_open** (*esp_http_client_handle_t* `client`, int `write_len`)

This function will be open the connection, write all header strings and return.

Return

- ESP_OK
- ESP_FAIL

Parameters

- [in] `client`: The `esp_http_client` handle
- [in] `write_len`: HTTP Content length need to write to the server

int **esp_http_client_write** (*esp_http_client_handle_t* `client`, const char *`buffer`, int `len`)

This function will write data to the HTTP connection previously opened by `esp_http_client_open()`

Return

- (-1) if any errors
- Length of data written

Parameters

- [in] `client`: The `esp_http_client` handle
- `buffer`: The buffer
- [in] `len`: This value must not be larger than the `write_len` parameter provided to `esp_http_client_open()`

int **esp_http_client_fetch_headers** (*esp_http_client_handle_t* `client`)

This function need to call after `esp_http_client_open`, it will read from http stream, process all receive headers.

Return

- (0) if stream doesn't contain content-length header, or chunked encoding (checked by `esp_http_client_is_chunked` response)
- (-1: ESP_FAIL) if any errors
- Download data length defined by content-length header

Parameters

- [in] `client`: The `esp_http_client` handle

bool **esp_http_client_is_chunked_response** (*esp_http_client_handle_t* `client`)

Check response data is chunked.

Return true or false

Parameters

- [in] `client`: The `esp_http_client` handle

int **esp_http_client_read** (*esp_http_client_handle_t* `client`, char *`buffer`, int `len`)

Read data from http stream.

Return

- (-1) if any errors
- Length of data was read

Parameters

- [in] `client`: The `esp_http_client` handle
- `buffer`: The buffer
- [in] `len`: The length

int **esp_http_client_get_status_code** (*esp_http_client_handle_t* `client`)

Get http response status code, the valid value if this function invoke after `esp_http_client_perform`

Return Status code

Parameters

- [in] `client`: The `esp_http_client` handle

int **esp_http_client_get_content_length** (*esp_http_client_handle_t client*)

Get http response content length (from header Content-Length) the valid value if this function invoke after `esp_http_client_perform`

Return

- (-1) Chunked transfer
- Content-Length value as bytes

Parameters

- [in] `client`: The `esp_http_client` handle

esp_err_t **esp_http_client_close** (*esp_http_client_handle_t client*)

Close http connection, still kept all http request resources.

Return

- ESP_OK
- ESP_FAIL

Parameters

- [in] `client`: The `esp_http_client` handle

esp_err_t **esp_http_client_cleanup** (*esp_http_client_handle_t client*)

This function must be the last function to call for an session. It is the opposite of the `esp_http_client_init` function and must be called with the same handle as input that a `esp_http_client_init` call returned. This might close all connections this handle has used and possibly has kept open until now. Don't call this function if you intend to transfer more files, re-using handles is a key to good performance with `esp_http_client`.

Return

- ESP_OK
- ESP_FAIL

Parameters

- [in] `client`: The `esp_http_client` handle

esp_http_client_transport_t **esp_http_client_get_transport_type** (*esp_http_client_handle_t client*)

Get transport type.

Return

- HTTP_TRANSPORT_UNKNOWN
- HTTP_TRANSPORT_OVER_TCP
- HTTP_TRANSPORT_OVER_SSL

Parameters

- [in] `client`: The `esp_http_client` handle

esp_err_t **esp_http_client_set_redirection** (*esp_http_client_handle_t client*)

Set redirection URL. When received the 30x code from the server, the client stores the redirect URL provided by the server. This function will set the current URL to redirect to enable client to execute the redirection request.

Return

- ESP_OK
- ESP_FAIL

Parameters

- [in] `client`: The `esp_http_client` handle

void **esp_http_client_add_auth** (*esp_http_client_handle_t client*)

On receiving HTTP Status code 401, this API can be invoked to add authorization information.

Note There is a possibility of receiving body message with redirection status codes, thus make sure to flush off body data after calling this API.

Parameters

- [in] `client`: The `esp_http_client` handle

bool **esp_http_client_is_complete_data_received** (*esp_http_client_handle_t client*)

Checks if entire data in the response has been read without any error.

Return

- true
- false

Parameters

- [in] `client`: The `esp_http_client` handle

Structures**struct esp_http_client_event**

HTTP Client events data.

Public Members*esp_http_client_event_id_t* **event_id**

event_id, to know the cause of the event

esp_http_client_handle_t **client**

esp_http_client_handle_t context

void ***data**

data of the event

int **data_len**

data length of data

void ***user_data**user_data context, from *esp_http_client_config_t* user_datachar ***header_key**

For HTTP_EVENT_ON_HEADER event_id, it' s store current http header key

char ***header_value**

For HTTP_EVENT_ON_HEADER event_id, it' s store current http header value

struct esp_http_client_config_t

HTTP configuration.

Public Members**const** char ***url**

HTTP URL, the information on the URL is most important, it overrides the other fields below, if any

const char ***host**

Domain or IP as string

int **port**

Port to connect, default depend on esp_http_client_transport_t (80 or 443)

const char ***username**

Using for Http authentication

const char ***password**

Using for Http authentication

esp_http_client_auth_type_t **auth_type**

Http authentication type, see esp_http_client_auth_type_t

const char ***path**

HTTP Path, if not set, default is /

const char ***query**

HTTP query

const char ***cert_pem**

SSL server certification, PEM format as string, if the client requires to verify server

const char *client_cert_pem
SSL client certification, PEM format as string, if the server requires to verify client

const char *client_key_pem
SSL client key, PEM format as string, if the server requires to verify client

const char *user_agent
The User Agent string to send with HTTP requests

esp_http_client_method_t **method**
HTTP Method

int timeout_ms
Network timeout in milliseconds

bool disable_auto_redirect
Disable HTTP automatic redirects

int max_redirection_count
Max redirection number, using default value if zero

http_event_handle_cb **event_handler**
HTTP Event Handle

esp_http_client_transport_t **transport_type**
HTTP transport type, see *esp_http_client_t* `transport_t`

int buffer_size
HTTP receive buffer size

int buffer_size_tx
HTTP transmit buffer size

void *user_data
HTTP user_data context

bool is_async
Set asynchronous mode, only supported with HTTPS for now

bool use_global_ca_store
Use a global ca_store for all the connections in which this bool is set.

bool skip_cert_common_name_check
Skip any validation of server certificate CN field

bool keep_alive_enable
Enable keep-alive timeout

int keep_alive_idle
Keep-alive idle time. Default is 5 (second)

int keep_alive_interval
Keep-alive interval time. Default is 5 (second)

int keep_alive_count
Keep-alive packet retry send count. Default is 3 counts

Macros

DEFAULT_HTTP_BUF_SIZE

ESP_ERR_HTTP_BASE
Starting number of HTTP error codes

ESP_ERR_HTTP_MAX_REDIRECT
The error exceeds the number of HTTP redirects

ESP_ERR_HTTP_CONNECT
Error open the HTTP connection

ESP_ERR_HTTP_WRITE_DATA

Error write HTTP data

ESP_ERR_HTTP_FETCH_HEADER

Error read HTTP header from server

ESP_ERR_HTTP_INVALID_TRANSPORT

There are no transport support for the input scheme

ESP_ERR_HTTP_CONNECTING

HTTP connection hasn't been established yet

ESP_ERR_HTTP_EAGAIN

Mapping of errno EAGAIN to esp_err_t

Type Definitions**typedef struct** esp_http_client ***esp_http_client_handle_t****typedef struct** esp_http_client_event ***esp_http_client_event_handle_t****typedef struct** esp_http_client_event **esp_http_client_event_t**

HTTP Client events data.

typedef esp_err_t (***http_event_handle_cb**)(esp_http_client_event_t *evt)**Enumerations****enum** esp_http_client_event_id_t

HTTP Client events id.

*Values:***HTTP_EVENT_ERROR** = 0

This event occurs when there are any errors during execution

HTTP_EVENT_ON_CONNECTED

Once the HTTP has been connected to the server, no data exchange has been performed

HTTP_EVENT_HEADERS_SENT

After sending all the headers to the server

HTTP_EVENT_HEADER_SENT = *HTTP_EVENT_HEADERS_SENT*

This header has been kept for backward compatibility and will be deprecated in future versions esp-idf

HTTP_EVENT_ON_HEADER

Occurs when receiving each header sent from the server

HTTP_EVENT_ON_DATA

Occurs when receiving data from the server, possibly multiple portions of the packet

HTTP_EVENT_ON_FINISH

Occurs when finish a HTTP session

HTTP_EVENT_DISCONNECTED

The connection has been disconnected

enum esp_http_client_transport_t

HTTP Client transport.

*Values:***HTTP_TRANSPORT_UNKNOWN** = 0x0

Unknown

HTTP_TRANSPORT_OVER_TCP

Transport over tcp

HTTP_TRANSPORT_OVER_SSL

Transport over ssl

enum esp_http_client_method_t

HTTP method.

Values:

HTTP_METHOD_GET = 0

HTTP GET Method

HTTP_METHOD_POST

HTTP POST Method

HTTP_METHOD_PUT

HTTP PUT Method

HTTP_METHOD_PATCH

HTTP PATCH Method

HTTP_METHOD_DELETE

HTTP DELETE Method

HTTP_METHOD_HEAD

HTTP HEAD Method

HTTP_METHOD_NOTIFY

HTTP NOTIFY Method

HTTP_METHOD_SUBSCRIBE

HTTP SUBSCRIBE Method

HTTP_METHOD_UNSUBSCRIBE

HTTP UNSUBSCRIBE Method

HTTP_METHOD_OPTIONS

HTTP OPTIONS Method

HTTP_METHOD_MAX

enum esp_http_client_auth_type_t

HTTP Authentication type.

Values:

HTTP_AUTH_TYPE_NONE = 0

No authentication

HTTP_AUTH_TYPE_BASIC

HTTP Basic authentication

HTTP_AUTH_TYPE_DIGEST

HTTP Digest authentication

enum HttpStatus_Code

Enum for the HTTP status codes.

Values:

HttpStatus_Ok = 200

HttpStatus_MultipleChoices = 300

HttpStatus_MovedPermanently = 301

HttpStatus_Found = 302

HttpStatus_TemporaryRedirect = 307

HttpStatus_Unauthorized = 401

2.4.5 HTTP Server

Overview

The HTTP Server component provides an ability for running a lightweight web server on ESP32. Following are detailed steps to use the API exposed by HTTP Server:

- `httpd_start()`: Creates an instance of HTTP server, allocate memory/resources for it depending upon the specified configuration and outputs a handle to the server instance. The server has both, a listening socket (TCP) for HTTP traffic, and a control socket (UDP) for control signals, which are selected in a round robin fashion in the server task loop. The task priority and stack size are configurable during server instance creation by passing `httpd_config_t` structure to `httpd_start()`. TCP traffic is parsed as HTTP requests and, depending on the requested URI, user registered handlers are invoked which are supposed to send back HTTP response packets.
- `httpd_stop()`: This stops the server with the provided handle and frees up any associated memory/resources. This is a blocking function that first signals a halt to the server task and then waits for the task to terminate. While stopping, the task will close all open connections, remove registered URI handlers and reset all session context data to empty.
- `httpd_register_uri_handler()`: A URI handler is registered by passing object of type `httpd_uri_t` structure which has members including uri name, method type (eg. HTTPD_GET/HTTPD_POST/HTTPD_PUT etc.), function pointer of type `esp_err_t *handler (httpd_req_t *req)` and `user_ctx` pointer to user context data.

Application Example

```

/* Our URI handler function to be called during GET /uri request */
esp_err_t get_handler(httpd_req_t *req)
{
    /* Send a simple response */
    const char resp[] = "URI GET Response";
    httpd_resp_send(req, resp, strlen(resp));
    return ESP_OK;
}

/* Our URI handler function to be called during POST /uri request */
esp_err_t post_handler(httpd_req_t *req)
{
    /* Destination buffer for content of HTTP POST request.
     * httpd_req_recv() accepts char* only, but content could
     * as well be any binary data (needs type casting).
     * In case of string data, null termination will be absent, and
     * content length would give length of string */
    char[100] content;

    /* Truncate if content length larger than the buffer */
    size_t recv_size = MIN(req->content_len, sizeof(content));

    int ret = httpd_req_recv(req, content, recv_size);
    if (ret <= 0) { /* 0 return value indicates connection closed */
        /* Check if timeout occurred */
        if (ret == HTTPD_SOCK_ERR_TIMEOUT) {
            /* In case of timeout one can choose to retry calling
             * httpd_req_recv(), but to keep it simple, here we
             * respond with an HTTP 408 (Request Timeout) error */
            httpd_resp_send_408(req);
        }
        /* In case of error, returning ESP_FAIL will
         * ensure that the underlying socket is closed */
        return ESP_FAIL;
    }
}

```

(continues on next page)

```
    }

    /* Send a simple response */
    const char resp[] = "URI POST Response";
    httpd_resp_send(req, resp, strlen(resp));
    return ESP_OK;
}

/* URI handler structure for GET /uri */
httpd_uri_t uri_get = {
    .uri      = "/uri",
    .method   = HTTP_GET,
    .handler  = get_handler,
    .user_ctx = NULL
};

/* URI handler structure for POST /uri */
httpd_uri_t uri_post = {
    .uri      = "/uri",
    .method   = HTTP_POST,
    .handler  = post_handler,
    .user_ctx = NULL
};

/* Function for starting the webserver */
httpd_handle_t start_webserver(void)
{
    /* Generate default configuration */
    httpd_config_t config = HTTPD_DEFAULT_CONFIG();

    /* Empty handle to esp_http_server */
    httpd_handle_t server = NULL;

    /* Start the httpd server */
    if (httpd_start(&server, &config) == ESP_OK) {
        /* Register URI handlers */
        httpd_register_uri_handler(server, &uri_get);
        httpd_register_uri_handler(server, &uri_post);
    }
    /* If server failed to start, handle will be NULL */
    return server;
}

/* Function for stopping the webserver */
void stop_webserver(httpd_handle_t server)
{
    if (server) {
        /* Stop the httpd server */
        httpd_stop(server);
    }
}
}
```

Simple HTTP server example Check HTTP server example under [protocols/http_server/simple](#) where handling of arbitrary content lengths, reading request headers and URL query parameters, and setting response headers is demonstrated.

Persistent Connections

HTTP server features persistent connections, allowing for the re-use of the same connection (session) for several transfers, all the while maintaining context specific data for the session. Context data may be allocated dynamically by the handler in which case a custom function may need to be specified for freeing this data when the connection/session is closed.

Persistent Connections Example

```

/* Custom function to free context */
void free_ctx_func(void *ctx)
{
    /* Could be something other than free */
    free(ctx);
}

esp_err_t adder_post_handler(httpd_req_t *req)
{
    /* Create session's context if not already available */
    if (! req->sess_ctx) {
        req->sess_ctx = malloc(sizeof(ANY_DATA_TYPE)); /*!< Pointer to context_
↳data */
        req->free_ctx = free_ctx_func; /*!< Function to free_
↳context data */
    }

    /* Access context data */
    ANY_DATA_TYPE *ctx_data = (ANY_DATA_TYPE *) req->sess_ctx;

    /* Respond */
    .....
    .....
    .....

    return ESP_OK;
}

```

Check the example under [protocols/http_server/persistent_sockets](#).

API Reference

Header File

- [esp_http_server/include/esp_http_server.h](#)

Functions

esp_err_t httpd_register_uri_handler (*httpd_handle_t* handle, **const** *httpd_uri_t* *uri_handler)
Registers a URI handler.

Example usage:

```

esp_err_t my_uri_handler(httpd_req_t* req)
{
    // Recv , Process and Send
    ....
    ....
    ....

    // Fail condition
    if (....) {

```

(continues on next page)

```

    // Return fail to close session //
    return ESP_FAIL;
}

// On success
return ESP_OK;
}

// URI handler structure
httpd_uri_t my_uri {
    .uri      = "/my_uri/path/xyz",
    .method   = HTTPD_GET,
    .handler  = my_uri_handler,
    .user_ctx = NULL
};

// Register handler
if (httpd_register_uri_handler(server_handle, &my_uri) != ESP_OK) {
    // If failed to register handler
    ....
}

```

Note URI handlers can be registered in real time as long as the server handle is valid.

Return

- ESP_OK : On successfully registering the handler
- ESP_ERR_INVALID_ARG : Null arguments
- ESP_ERR_HTTPD_HANDLERS_FULL : If no slots left for new handler
- ESP_ERR_HTTPD_HANDLER_EXISTS : If handler with same URI and method is already registered

Parameters

- [in] handle: handle to HTTPD server instance
- [in] uri_handler: pointer to handler that needs to be registered

esp_err_t **httpd_unregister_uri_handler** (*httpd_handle_t* handle, **const** char *uri, *httpd_method_t* method)

Unregister a URI handler.

Return

- ESP_OK : On successfully deregistering the handler
- ESP_ERR_INVALID_ARG : Null arguments
- ESP_ERR_NOT_FOUND : Handler with specified URI and method not found

Parameters

- [in] handle: handle to HTTPD server instance
- [in] uri: URI string
- [in] method: HTTP method

esp_err_t **httpd_unregister_uri** (*httpd_handle_t* handle, **const** char *uri)

Unregister all URI handlers with the specified uri string.

Return

- ESP_OK : On successfully deregistering all such handlers
- ESP_ERR_INVALID_ARG : Null arguments
- ESP_ERR_NOT_FOUND : No handler registered with specified uri string

Parameters

- [in] handle: handle to HTTPD server instance
- [in] uri: uri string specifying all handlers that need to be deregistered

esp_err_t **httpd_sess_set_recv_override** (*httpd_handle_t* hd, int sockfd, *httpd_recv_func_t* recv_func)

Override web server' s receive function (by session FD)

This function overrides the web server' s receive function. This same function is used to read HTTP request

packets.

Note This API is supposed to be called either from the context of

- an http session APIs where sockfd is a valid parameter
- a URI handler where sockfd is obtained using `httpd_req_to_sockfd()`

Return

- `ESP_OK` : On successfully registering override
- `ESP_ERR_INVALID_ARG` : Null arguments

Parameters

- `[in]` `hd`: HTTPD instance handle
- `[in]` `sockfd`: Session socket FD
- `[in]` `recv_func`: The receive function to be set for this session

`esp_err_t httpd_sess_set_send_override` (`httpd_handle_t` *hd*, `int` *sockfd*, `httpd_send_func_t` *send_func*)

Override web server' s send function (by session FD)

This function overrides the web server' s send function. This same function is used to send out any response to any HTTP request.

Note This API is supposed to be called either from the context of

- an http session APIs where sockfd is a valid parameter
- a URI handler where sockfd is obtained using `httpd_req_to_sockfd()`

Return

- `ESP_OK` : On successfully registering override
- `ESP_ERR_INVALID_ARG` : Null arguments

Parameters

- `[in]` `hd`: HTTPD instance handle
- `[in]` `sockfd`: Session socket FD
- `[in]` `send_func`: The send function to be set for this session

`esp_err_t httpd_sess_set_pending_override` (`httpd_handle_t` *hd*, `int` *sockfd*, `httpd_pending_func_t` *pending_func*)

Override web server' s pending function (by session FD)

This function overrides the web server' s pending function. This function is used to test for pending bytes in a socket.

Note This API is supposed to be called either from the context of

- an http session APIs where sockfd is a valid parameter
- a URI handler where sockfd is obtained using `httpd_req_to_sockfd()`

Return

- `ESP_OK` : On successfully registering override
- `ESP_ERR_INVALID_ARG` : Null arguments

Parameters

- `[in]` `hd`: HTTPD instance handle
- `[in]` `sockfd`: Session socket FD
- `[in]` `pending_func`: The receive function to be set for this session

`int httpd_req_to_sockfd` (`httpd_req_t` **r*)

Get the Socket Descriptor from the HTTP request.

This API will return the socket descriptor of the session for which URI handler was executed on reception of HTTP request. This is useful when user wants to call functions that require session socket fd, from within a URI handler, ie. : `httpd_sess_get_ctx()`, `httpd_sess_trigger_close()`, `httpd_sess_update_lru_counter()`.

Note This API is supposed to be called only from the context of a URI handler where `httpd_req_t*` request pointer is valid.

Return

- Socket descriptor : The socket descriptor for this request
- -1 : Invalid/NULL request pointer

Parameters

- `[in]` `r`: The request whose socket descriptor should be found

int **httpd_req_recv** (*httpd_req_t* *r, char *buf, size_t buf_len)

API to read content data from the HTTP request.

This API will read HTTP content data from the HTTP request into provided buffer. Use content_len provided in httpd_req_t structure to know the length of data to be fetched. If content_len is too large for the buffer then user may have to make multiple calls to this function, each time fetching 'buf_len' number of bytes, while the pointer to content data is incremented internally by the same number.

Note

- This API is supposed to be called only from the context of a URI handler where httpd_req_t* request pointer is valid.
- If an error is returned, the URI handler must further return an error. This will ensure that the erroneous socket is closed and cleaned up by the web server.
- Presently Chunked Encoding is not supported

Return

- Bytes : Number of bytes read into the buffer successfully
- 0 : Buffer length parameter is zero / connection closed by peer
- HTTPD_SOCK_ERR_INVALID : Invalid arguments
- HTTPD_SOCK_ERR_TIMEOUT : Timeout/interrupted while calling socket recv()
- HTTPD_SOCK_ERR_FAIL : Unrecoverable error while calling socket recv()

Parameters

- [in] r: The request being responded to
- [in] buf: Pointer to a buffer that the data will be read into
- [in] buf_len: Length of the buffer

size_t **httpd_req_get_hdr_value_len** (*httpd_req_t* *r, const char *field)

Search for a field in request headers and return the string length of it' s value.

Note

- This API is supposed to be called only from the context of a URI handler where httpd_req_t* request pointer is valid.
- Once httpd_resp_send() API is called all request headers are purged, so request headers need be copied into separate buffers if they are required later.

Return

- Length : If field is found in the request URL
- Zero : Field not found / Invalid request / Null arguments

Parameters

- [in] r: The request being responded to
- [in] field: The header field to be searched in the request

esp_err_t **httpd_req_get_hdr_value_str** (*httpd_req_t* *r, const char *field, char *val, size_t *val_size*)

Get the value string of a field from the request headers.

Note

- This API is supposed to be called only from the context of a URI handler where httpd_req_t* request pointer is valid.
- Once httpd_resp_send() API is called all request headers are purged, so request headers need be copied into separate buffers if they are required later.
- If output size is greater than input, then the value is truncated, accompanied by truncation error as return value.
- Use httpd_req_get_hdr_value_len() to know the right buffer length

Return

- ESP_OK : Field found in the request header and value string copied
- ESP_ERR_NOT_FOUND : Key not found
- ESP_ERR_INVALID_ARG : Null arguments
- ESP_ERR_HTTPD_INVALID_REQ : Invalid HTTP request pointer
- ESP_ERR_HTTPD_RESULT_TRUNC : Value string truncated

Parameters

- [in] r: The request being responded to
- [in] field: The field to be searched in the header

- [out] val: Pointer to the buffer into which the value will be copied if the field is found
- [in] val_size: Size of the user buffer “val”

size_t **httpd_req_get_url_query_len** (*httpd_req_t* *r)

Get Query string length from the request URL.

Note This API is supposed to be called only from the context of a URI handler where *httpd_req_t** request pointer is valid

Return

- Length : Query is found in the request URL
- Zero : Query not found / Null arguments / Invalid request

Parameters

- [in] r: The request being responded to

esp_err_t **httpd_req_get_url_query_str** (*httpd_req_t* *r, char *buf, size_t buf_len)

Get Query string from the request URL.

Note

- Presently, the user can fetch the full URL query string, but decoding will have to be performed by the user. Request headers can be read using *httpd_req_get_hdr_value_str()* to know the ‘Content-Type’ (eg. Content-Type: application/x-www-form-urlencoded) and then the appropriate decoding algorithm needs to be applied.
- This API is supposed to be called only from the context of a URI handler where *httpd_req_t** request pointer is valid
- If output size is greater than input, then the value is truncated, accompanied by truncation error as return value
- Prior to calling this function, one can use *httpd_req_get_url_query_len()* to know the query string length beforehand and hence allocate the buffer of right size (usually query string length + 1 for null termination) for storing the query string

Return

- ESP_OK : Query is found in the request URL and copied to buffer
- ESP_ERR_NOT_FOUND : Query not found
- ESP_ERR_INVALID_ARG : Null arguments
- ESP_ERR_HTTPD_INVALID_REQ : Invalid HTTP request pointer
- ESP_ERR_HTTPD_RESULT_TRUNC : Query string truncated

Parameters

- [in] r: The request being responded to
- [out] buf: Pointer to the buffer into which the query string will be copied (if found)
- [in] buf_len: Length of output buffer

esp_err_t **httpd_query_key_value** (const char *qry, const char *key, char *val, size_t val_size)

Helper function to get a URL query tag from a query string of the type param1=val1¶m2=val2.

Note

- The components of URL query string (keys and values) are not URLdecoded. The user must check for ‘Content-Type’ field in the request headers and then depending upon the specified encoding (URLencoded or otherwise) apply the appropriate decoding algorithm.
- If actual value size is greater than val_size, then the value is truncated, accompanied by truncation error as return value.

Return

- ESP_OK : Key is found in the URL query string and copied to buffer
- ESP_ERR_NOT_FOUND : Key not found
- ESP_ERR_INVALID_ARG : Null arguments
- ESP_ERR_HTTPD_RESULT_TRUNC : Value string truncated

Parameters

- [in] qry: Pointer to query string
- [in] key: The key to be searched in the query string
- [out] val: Pointer to the buffer into which the value will be copied if the key is found
- [in] val_size: Size of the user buffer “val”

bool **httpd_uri_match_wildcard** (const char *uri_template, const char *uri_to_match, size_t match_upto)

Test if a URI matches the given wildcard template.

Template may end with “?” to make the previous character optional (typically a slash), “*” for a wildcard match, and “?*” to make the previous character optional, and if present, allow anything to follow.

Example:

- * matches everything
- /foo/? matches /foo and /foo/
- /foo/* (sans the backslash) matches /foo/ and /foo/bar, but not /foo or /fo
- /foo/?* or /foo/*? (sans the backslash) matches /foo/, /foo/bar, and also /foo, but not /foox or /fo

The special characters “?” and “*” anywhere else in the template will be taken literally.

Return true if a match was found

Parameters

- [in] uri_template: URI template (pattern)
- [in] uri_to_match: URI to be matched
- [in] match_upto: how many characters of the URI buffer to test (there may be trailing query string etc.)

esp_err_t **httpd_resp_send** (*httpd_req_t* *r, const char *buf, ssize_t buf_len)

API to send a complete HTTP response.

This API will send the data as an HTTP response to the request. This assumes that you have the entire response ready in a single buffer. If you wish to send response in incremental chunks use `httpd_resp_send_chunk()` instead.

If no status code and content-type were set, by default this will send 200 OK status code and content type as text/html. You may call the following functions before this API to configure the response headers : `httpd_resp_set_status()` - for setting the HTTP status string, `httpd_resp_set_type()` - for setting the Content Type, `httpd_resp_set_hdr()` - for appending any additional field value entries in the response header

Note

- This API is supposed to be called only from the context of a URI handler where `httpd_req_t*` request pointer is valid.
- Once this API is called, the request has been responded to.
- No additional data can then be sent for the request.
- Once this API is called, all request headers are purged, so request headers need be copied into separate buffers if they are required later.

Return

- ESP_OK : On successfully sending the response packet
- ESP_ERR_INVALID_ARG : Null request pointer
- ESP_ERR_HTTPD_RESP_HDR : Essential headers are too large for internal buffer
- ESP_ERR_HTTPD_RESP_SEND : Error in raw send
- ESP_ERR_HTTPD_INVALID_REQ : Invalid request

Parameters

- [in] r: The request being responded to
- [in] buf: Buffer from where the content is to be fetched
- [in] buf_len: Length of the buffer, HTTPD_RESP_USE_STRLEN to use `strlen()`

esp_err_t **httpd_resp_send_chunk** (*httpd_req_t* *r, const char *buf, ssize_t buf_len)

API to send one HTTP chunk.

This API will send the data as an HTTP response to the request. This API will use chunked-encoding and send the response in the form of chunks. If you have the entire response contained in a single buffer, please use `httpd_resp_send()` instead.

If no status code and content-type were set, by default this will send 200 OK status code and content type as text/html. You may call the following functions before this API to configure the response headers `httpd_resp_set_status()` - for setting the HTTP status string, `httpd_resp_set_type()` - for setting the Content Type, `httpd_resp_set_hdr()` - for appending any additional field value entries in the response header

Note

- This API is supposed to be called only from the context of a URI handler where `httpd_req_t*` request pointer is valid.
- When you are finished sending all your chunks, you must call this function with `buf_len` as 0.
- Once this API is called, all request headers are purged, so request headers need be copied into separate buffers if they are required later.

Return

- `ESP_OK` : On successfully sending the response packet chunk
- `ESP_ERR_INVALID_ARG` : Null request pointer
- `ESP_ERR_HTTPD_RESP_HDR` : Essential headers are too large for internal buffer
- `ESP_ERR_HTTPD_RESP_SEND` : Error in raw send
- `ESP_ERR_HTTPD_INVALID_REQ` : Invalid request pointer

Parameters

- `[in]` `r`: The request being responded to
- `[in]` `buf`: Pointer to a buffer that stores the data
- `[in]` `buf_len`: Length of the buffer, `HTTPD_RESP_USE_STRLEN` to use `strlen()`

static `esp_err_t httpd_resp_sendstr` (`httpd_req_t` **r*, **const** char **str*)

API to send a complete string as HTTP response.

This API simply calls `http_resp_send` with buffer length set to string length assuming the buffer contains a null terminated string

Return

- `ESP_OK` : On successfully sending the response packet
- `ESP_ERR_INVALID_ARG` : Null request pointer
- `ESP_ERR_HTTPD_RESP_HDR` : Essential headers are too large for internal buffer
- `ESP_ERR_HTTPD_RESP_SEND` : Error in raw send
- `ESP_ERR_HTTPD_INVALID_REQ` : Invalid request

Parameters

- `[in]` `r`: The request being responded to
- `[in]` `str`: String to be sent as response body

static `esp_err_t httpd_resp_sendstr_chunk` (`httpd_req_t` **r*, **const** char **str*)

API to send a string as an HTTP response chunk.

This API simply calls `http_resp_send_chunk` with buffer length set to string length assuming the buffer contains a null terminated string

Return

- `ESP_OK` : On successfully sending the response packet
- `ESP_ERR_INVALID_ARG` : Null request pointer
- `ESP_ERR_HTTPD_RESP_HDR` : Essential headers are too large for internal buffer
- `ESP_ERR_HTTPD_RESP_SEND` : Error in raw send
- `ESP_ERR_HTTPD_INVALID_REQ` : Invalid request

Parameters

- `[in]` `r`: The request being responded to
- `[in]` `str`: String to be sent as response body (NULL to finish response packet)

`esp_err_t httpd_resp_set_status` (`httpd_req_t` **r*, **const** char **status*)

API to set the HTTP status code.

This API sets the status of the HTTP response to the value specified. By default, the ‘200 OK’ response is sent as the response.

Note

- This API is supposed to be called only from the context of a URI handler where `httpd_req_t*` request pointer is valid.
- This API only sets the status to this value. The status isn’ t sent out until any of the send APIs is executed.
- Make sure that the lifetime of the status string is valid till send function is called.

Return

- `ESP_OK` : On success
- `ESP_ERR_INVALID_ARG` : Null arguments

- `ESP_ERR_HTTPD_INVALID_REQ` : Invalid request pointer

Parameters

- `[in]` `r`: The request being responded to
- `[in]` `status`: The HTTP status code of this response

esp_err_t **httpd_resp_set_type** (*httpd_req_t* **r*, **const** char **type*)

API to set the HTTP content type.

This API sets the ‘Content Type’ field of the response. The default content type is ‘text/html’ .

Note

- This API is supposed to be called only from the context of a URI handler where `httpd_req_t*` request pointer is valid.
- This API only sets the content type to this value. The type isn’ t sent out until any of the send APIs is executed.
- Make sure that the lifetime of the type string is valid till send function is called.

Return

- `ESP_OK` : On success
- `ESP_ERR_INVALID_ARG` : Null arguments
- `ESP_ERR_HTTPD_INVALID_REQ` : Invalid request pointer

Parameters

- `[in]` `r`: The request being responded to
- `[in]` `type`: The Content Type of the response

esp_err_t **httpd_resp_set_hdr** (*httpd_req_t* **r*, **const** char **field*, **const** char **value*)

API to append any additional headers.

This API sets any additional header fields that need to be sent in the response.

Note

- This API is supposed to be called only from the context of a URI handler where `httpd_req_t*` request pointer is valid.
- The header isn’ t sent out until any of the send APIs is executed.
- The maximum allowed number of additional headers is limited to value of `max_resp_headers` in config structure.
- Make sure that the lifetime of the field value strings are valid till send function is called.

Return

- `ESP_OK` : On successfully appending new header
- `ESP_ERR_INVALID_ARG` : Null arguments
- `ESP_ERR_HTTPD_RESP_HDR` : Total additional headers exceed max allowed
- `ESP_ERR_HTTPD_INVALID_REQ` : Invalid request pointer

Parameters

- `[in]` `r`: The request being responded to
- `[in]` `field`: The field name of the HTTP header
- `[in]` `value`: The value of this HTTP header

esp_err_t **httpd_resp_send_err** (*httpd_req_t* **req*, *httpd_err_code_t* *error*, **const** char **msg*)

For sending out error code in response to HTTP request.

Note

- This API is supposed to be called only from the context of a URI handler where `httpd_req_t*` request pointer is valid.
- Once this API is called, all request headers are purged, so request headers need be copied into separate buffers if they are required later.
- If you wish to send additional data in the body of the response, please use the lower-level functions directly.

Return

- `ESP_OK` : On successfully sending the response packet
- `ESP_ERR_INVALID_ARG` : Null arguments
- `ESP_ERR_HTTPD_RESP_SEND` : Error in raw send
- `ESP_ERR_HTTPD_INVALID_REQ` : Invalid request pointer

Parameters

- [in] req: Pointer to the HTTP request for which the response needs to be sent
- [in] error: Error type to send
- [in] msg: Error message string (pass NULL for default message)

static *esp_err_t* **httpd_resp_send_404** (*httpd_req_t* *r)

Helper function for HTTP 404.

Send HTTP 404 message. If you wish to send additional data in the body of the response, please use the lower-level functions directly.

Note

- This API is supposed to be called only from the context of a URI handler where *httpd_req_t** request pointer is valid.
- Once this API is called, all request headers are purged, so request headers need be copied into separate buffers if they are required later.

Return

- ESP_OK : On successfully sending the response packet
- ESP_ERR_INVALID_ARG : Null arguments
- ESP_ERR_HTTPD_RESP_SEND : Error in raw send
- ESP_ERR_HTTPD_INVALID_REQ : Invalid request pointer

Parameters

- [in] r: The request being responded to

static *esp_err_t* **httpd_resp_send_408** (*httpd_req_t* *r)

Helper function for HTTP 408.

Send HTTP 408 message. If you wish to send additional data in the body of the response, please use the lower-level functions directly.

Note

- This API is supposed to be called only from the context of a URI handler where *httpd_req_t** request pointer is valid.
- Once this API is called, all request headers are purged, so request headers need be copied into separate buffers if they are required later.

Return

- ESP_OK : On successfully sending the response packet
- ESP_ERR_INVALID_ARG : Null arguments
- ESP_ERR_HTTPD_RESP_SEND : Error in raw send
- ESP_ERR_HTTPD_INVALID_REQ : Invalid request pointer

Parameters

- [in] r: The request being responded to

static *esp_err_t* **httpd_resp_send_500** (*httpd_req_t* *r)

Helper function for HTTP 500.

Send HTTP 500 message. If you wish to send additional data in the body of the response, please use the lower-level functions directly.

Note

- This API is supposed to be called only from the context of a URI handler where *httpd_req_t** request pointer is valid.
- Once this API is called, all request headers are purged, so request headers need be copied into separate buffers if they are required later.

Return

- ESP_OK : On successfully sending the response packet
- ESP_ERR_INVALID_ARG : Null arguments
- ESP_ERR_HTTPD_RESP_SEND : Error in raw send
- ESP_ERR_HTTPD_INVALID_REQ : Invalid request pointer

Parameters

- [in] r: The request being responded to

int **httpd_send** (*httpd_req_t* *r, const char *buf, size_t buf_len)

Raw HTTP send.

Call this API if you wish to construct your custom response packet. When using this, all essential header, eg. HTTP version, Status Code, Content Type and Length, Encoding, etc. will have to be constructed manually, and HTTP delimiters (CRLF) will need to be placed correctly for separating sub-sections of the HTTP response packet.

If the send override function is set, this API will end up calling that function eventually to send data out.

Note

- This API is supposed to be called only from the context of a URI handler where `httpd_req_t*` request pointer is valid.
- Unless the response has the correct HTTP structure (which the user must now ensure) it is not guaranteed that it will be recognized by the client. For most cases, you wouldn't have to call this API, but you would rather use either of : `httpd_resp_send()`, `httpd_resp_send_chunk()`

Return

- Bytes : Number of bytes that were sent successfully
- `HTTPD_SOCK_ERR_INVALID` : Invalid arguments
- `HTTPD_SOCK_ERR_TIMEOUT` : Timeout/interrupted while calling socket send()
- `HTTPD_SOCK_ERR_FAIL` : Unrecoverable error while calling socket send()

Parameters

- [in] `r`: The request being responded to
- [in] `buf`: Buffer from where the fully constructed packet is to be read
- [in] `buf_len`: Length of the buffer

```
esp_err_t httpd_register_err_handler(httpd_handle_t handle, httpd_err_code_t error,
                                     httpd_err_handler_func_t handler_fn)
```

Function for registering HTTP error handlers.

This function maps a handler function to any supported error code given by `httpd_err_code_t`. See prototype `httpd_err_handler_func_t` above for details.

Return

- `ESP_OK` : handler registered successfully
- `ESP_ERR_INVALID_ARG` : invalid error code or server handle

Parameters

- [in] `handle`: HTTP server handle
- [in] `error`: Error type
- [in] `handler_fn`: User implemented handler function (Pass NULL to unset any previously set handler)

```
esp_err_t httpd_start(httpd_handle_t*handle, const httpd_config_t*config)
```

Starts the web server.

Create an instance of HTTP server and allocate memory/resources for it depending upon the specified configuration.

Example usage:

```
//Function for starting the webserver
httpd_handle_t start_webserver(void)
{
    // Generate default configuration
    httpd_config_t config = HTTPD_DEFAULT_CONFIG();

    // Empty handle to http_server
    httpd_handle_t server = NULL;

    // Start the httpd server
    if (httpd_start(&server, &config) == ESP_OK) {
        // Register URI handlers
        httpd_register_uri_handler(server, &uri_get);
        httpd_register_uri_handler(server, &uri_post);
    }
    // If server failed to start, handle will be NULL
}
```

(continues on next page)


```

return server;
}

```

Return

- ESP_OK : Instance created successfully
- ESP_ERR_INVALID_ARG : Null argument(s)
- ESP_ERR_HTTPD_ALLOC_MEM : Failed to allocate memory for instance
- ESP_ERR_HTTPD_TASK : Failed to launch server task

Parameters

- [in] config: Configuration for new instance of the server
- [out] handle: Handle to newly created instance of the server. NULL on error

esp_err_t **httpd_stop** (*httpd_handle_t* handle)

Stops the web server.

Deallocates memory/resources used by an HTTP server instance and deletes it. Once deleted the handle can no longer be used for accessing the instance.

Example usage:

```

// Function for stopping the webserver
void stop_webserver(httpd_handle_t server)
{
    // Ensure handle is non NULL
    if (server != NULL) {
        // Stop the httpd server
        httpd_stop(server);
    }
}

```

Return

- ESP_OK : Server stopped successfully
- ESP_ERR_INVALID_ARG : Handle argument is Null

Parameters

- [in] handle: Handle to server returned by httpd_start

esp_err_t **httpd_queue_work** (*httpd_handle_t* handle, *httpd_work_fn_t* work, void *arg)

Queue execution of a function in HTTPD's context.

This API queues a work function for asynchronous execution

Note Some protocols require that the web server generate some asynchronous data and send it to the persistently opened connection. This facility is for use by such protocols.

Return

- ESP_OK : On successfully queueing the work
- ESP_FAIL : Failure in ctrl socket
- ESP_ERR_INVALID_ARG : Null arguments

Parameters

- [in] handle: Handle to server returned by httpd_start
- [in] work: Pointer to the function to be executed in the HTTPD's context
- [in] arg: Pointer to the arguments that should be passed to this function

void ***httpd_sess_get_ctx** (*httpd_handle_t* handle, int sockfd)

Get session context from socket descriptor.

Typically if a session context is created, it is available to URI handlers through the httpd_req_t structure. But, there are cases where the web server's send/receive functions may require the context (for example, for accessing keying information etc). Since the send/receive function only have the socket descriptor at their disposal, this API provides them with a way to retrieve the session context.

Return

- void* : Pointer to the context associated with this session

- NULL : Empty context / Invalid handle / Invalid socket fd

Parameters

- [in] handle: Handle to server returned by `httpd_start`
- [in] sockfd: The socket descriptor for which the context should be extracted.

void `httpd_sess_set_ctx` (*httpd_handle_t* handle, int sockfd, void *ctx, *httpd_free_ctx_fn_t* free_fn)
Set session context by socket descriptor.

Parameters

- [in] handle: Handle to server returned by `httpd_start`
- [in] sockfd: The socket descriptor for which the context should be extracted.
- [in] ctx: Context object to assign to the session
- [in] free_fn: Function that should be called to free the context

void *`httpd_sess_get_transport_ctx` (*httpd_handle_t* handle, int sockfd)
Get session 'transport' context by socket descriptor.

This context is used by the send/receive functions, for example to manage SSL context.

See `httpd_sess_get_ctx()`

Return

- void* : Pointer to the transport context associated with this session
- NULL : Empty context / Invalid handle / Invalid socket fd

Parameters

- [in] handle: Handle to server returned by `httpd_start`
- [in] sockfd: The socket descriptor for which the context should be extracted.

void `httpd_sess_set_transport_ctx` (*httpd_handle_t* handle, int sockfd, void *ctx, *httpd_free_ctx_fn_t* free_fn)
Set session 'transport' context by socket descriptor.

See `httpd_sess_set_ctx()`

Parameters

- [in] handle: Handle to server returned by `httpd_start`
- [in] sockfd: The socket descriptor for which the context should be extracted.
- [in] ctx: Transport context object to assign to the session
- [in] free_fn: Function that should be called to free the transport context

void *`httpd_get_global_user_ctx` (*httpd_handle_t* handle)
Get HTTPD global user context (it was set in the server config struct)

Return global user context

Parameters

- [in] handle: Handle to server returned by `httpd_start`

void *`httpd_get_global_transport_ctx` (*httpd_handle_t* handle)
Get HTTPD global transport context (it was set in the server config struct)

Return global transport context

Parameters

- [in] handle: Handle to server returned by `httpd_start`

esp_err_t `httpd_sess_trigger_close` (*httpd_handle_t* handle, int sockfd)
Trigger an httpd session close externally.

Note Calling this API is only required in special circumstances wherein some application requires to close an httpd client session asynchronously.

Return

- ESP_OK : On successfully initiating closure
- ESP_FAIL : Failure to queue work
- ESP_ERR_NOT_FOUND : Socket fd not found
- ESP_ERR_INVALID_ARG : Null arguments

Parameters

- [in] handle: Handle to server returned by `httpd_start`

- [in] `sockfd`: The socket descriptor of the session to be closed

`esp_err_t httpd_sess_update_lru_counter` (`httpd_handle_t handle`, int `sockfd`)

Update LRU counter for a given socket.

LRU Counters are internally associated with each session to monitor how recently a session exchanged traffic. When LRU purge is enabled, if a client is requesting for connection but maximum number of sockets/sessions is reached, then the session having the earliest LRU counter is closed automatically.

Updating the LRU counter manually prevents the socket from being purged due to the Least Recently Used (LRU) logic, even though it might not have received traffic for some time. This is useful when all open sockets/session are frequently exchanging traffic but the user specifically wants one of the sessions to be kept open, irrespective of when it last exchanged a packet.

Note Calling this API is only necessary if the LRU Purge Enable option is enabled.

Return

- `ESP_OK` : Socket found and LRU counter updated
- `ESP_ERR_NOT_FOUND` : Socket not found
- `ESP_ERR_INVALID_ARG` : Null arguments

Parameters

- [in] `handle`: Handle to server returned by `httpd_start`
- [in] `sockfd`: The socket descriptor of the session for which LRU counter is to be updated

Structures

`struct httpd_config`

HTTP Server Configuration Structure.

Note Use `HTTPD_DEFAULT_CONFIG()` to initialize the configuration to a default value and then modify only those fields that are specifically determined by the use case.

Public Members

unsigned `task_priority`

Priority of FreeRTOS task which runs the server

size_t `stack_size`

The maximum stack size allowed for the server task

BaseType_t `core_id`

The core the HTTP server task will run on

uint16_t `server_port`

TCP Port number for receiving and transmitting HTTP traffic

uint16_t `ctrl_port`

UDP Port number for asynchronously exchanging control signals between various components of the server

uint16_t `max_open_sockets`

Max number of sockets/clients connected at any time

uint16_t `max_uri_handlers`

Maximum allowed uri handlers

uint16_t `max_resp_headers`

Maximum allowed additional headers in HTTP response

uint16_t `backlog_conn`

Number of backlog connections

bool `lru_purge_enable`

Purge “Least Recently Used” connection

uint16_t `recv_wait_timeout`
Timeout for `recv` function (in seconds)

uint16_t `send_wait_timeout`
Timeout for `send` function (in seconds)

void *`global_user_ctx`
Global user context.

This field can be used to store arbitrary user data within the server context. The value can be retrieved using the server handle, available e.g. in the `httpd_req_t` struct.

When shutting down, the server frees up the user context by calling `free()` on the `global_user_ctx` field. If you wish to use a custom function for freeing the global user context, please specify that here.

[*httpd_free_ctx_fn_t*](#) `global_user_ctx_free_fn`
Free function for global user context

void *`global_transport_ctx`
Global transport context.

Similar to `global_user_ctx`, but used for session encoding or encryption (e.g. to hold the SSL context). It will be freed using `free()`, unless `global_transport_ctx_free_fn` is specified.

[*httpd_free_ctx_fn_t*](#) `global_transport_ctx_free_fn`
Free function for global transport context

[*httpd_open_func_t*](#) `open_fn`
Custom session opening callback.

Called on a new session socket just after `accept()`, but before reading any data.

This is an opportunity to set up e.g. SSL encryption using `global_transport_ctx` and the `send/recv/pending` session overrides.

If a context needs to be maintained between these functions, store it in the session using `httpd_sess_set_transport_ctx()` and retrieve it later with `httpd_sess_get_transport_ctx()`

Returning a value other than `ESP_OK` will immediately close the new socket.

[*httpd_close_func_t*](#) `close_fn`
Custom session closing callback.

Called when a session is deleted, before freeing user and transport contexts and before closing the socket. This is a place for custom de-init code common to all sockets.

Set the user or transport context to `NULL` if it was freed here, so the server does not try to free it again.

This function is run for all terminated sessions, including sessions where the socket was closed by the network stack - that is, the file descriptor may not be valid anymore.

[*httpd_uri_match_func_t*](#) `uri_match_fn`
URI matcher function.

Called when searching for a matching URI: 1) whose request handler is to be executed right after an HTTP request is successfully parsed 2) in order to prevent duplication while registering a new URI handler using `httpd_register_uri_handler()`

Available options are: 1) `NULL` : Internally do basic matching using `strncmp()` 2) `httpd_uri_match_wildcard()` : URI wildcard matcher

Users can implement their own matching functions (See description of the `httpd_uri_match_func_t` function prototype)

struct `httpd_req`
HTTP Request Data Structure.

Public Members

httpd_handle_t **handle**

Handle to server instance

int method

The type of HTTP request, -1 if unsupported method

const char uri[HTTPD_MAX_URI_LEN + 1]

The URI of this request (1 byte extra for null termination)

size_t content_len

Length of the request body

void *aux

Internally used members

void *user_ctx

User context pointer passed during URI registration.

void *sess_ctx

Session Context Pointer

A session context. Contexts are maintained across ‘sessions’ for a given open TCP connection. One session could have multiple request responses. The web server will ensure that the context persists across all these request and responses.

By default, this is NULL. URI Handlers can set this to any meaningful value.

If the underlying socket gets closed, and this pointer is non-NULL, the web server will free up the context by calling `free()`, unless `free_ctx` function is set.

httpd_free_ctx_fn_t **free_ctx**

Pointer to free context hook

Function to free session context

If the web server’s socket closes, it frees up the session context by calling `free()` on the `sess_ctx` member. If you wish to use a custom function for freeing the session context, please specify that here.

bool ignore_sess_ctx_changes

Flag indicating if Session Context changes should be ignored

By default, if you change the `sess_ctx` in some URI handler, the http server will internally free the earlier context (if non NULL), after the URI handler returns. If you want to manage the allocation/reallocation/freeing of `sess_ctx` yourself, set this flag to true, so that the server will not perform any checks on it. The context will be cleared by the server (by calling `free_ctx` or `free()`) only if the socket gets closed.

struct httpd_uri

Structure for URI handler.

Public Members

const char *uri

The URI to handle

httpd_method_t **method**

Method supported by the URI

esp_err_t (***handler**) (*httpd_req_t* *r)

Handler to call for supported request method. This must return `ESP_OK`, or else the underlying socket will be closed.

void *user_ctx

Pointer to user context data which will be available to handler

Macros**HTTPD_MAX_REQ_HDR_LEN****HTTPD_MAX_URI_LEN****HTTPD SOCK_ERR_FAIL****HTTPD SOCK_ERR_INVALID****HTTPD SOCK_ERR_TIMEOUT****HTTPD_200**

HTTP Response 200

HTTPD_204

HTTP Response 204

HTTPD_207

HTTP Response 207

HTTPD_400

HTTP Response 400

HTTPD_404

HTTP Response 404

HTTPD_408

HTTP Response 408

HTTPD_500

HTTP Response 500

HTTPD_TYPE_JSON

HTTP Content type JSON

HTTPD_TYPE_TEXT

HTTP Content type text/HTML

HTTPD_TYPE_OCTET

HTTP Content type octext-stream

HTTPD_DEFAULT_CONFIG ()**ESP_ERR_HTTPD_BASE**

Starting number of HTTPD error codes

ESP_ERR_HTTPD_HANDLERS_FULL

All slots for registering URI handlers have been consumed

ESP_ERR_HTTPD_HANDLER_EXISTS

URI handler with same method and target URI already registered

ESP_ERR_HTTPD_INVALID_REQ

Invalid request pointer

ESP_ERR_HTTPD_RESULT_TRUNC

Result string truncated

ESP_ERR_HTTPD_RESP_HDR

Response header field larger than supported

ESP_ERR_HTTPD_RESP_SEND

Error occurred while sending response packet

ESP_ERR_HTTPD_ALLOC_MEM

Failed to dynamically allocate memory for resource

ESP_ERR_HTTPD_TASK

Failed to launch server task/thread

HTTPD_RESP_USE_STRLEN

Type Definitions

typedef struct *httpd_req* httpd_req_t

HTTP Request Data Structure.

typedef struct *httpd_uri* httpd_uri_t

Structure for URI handler.

typedef int (*httpd_send_func_t) (*httpd_handle_t* hd, int sockfd, const char *buf, size_t buf_len, int flags)

Prototype for HTTPDs low-level send function.

Note User specified send function must handle errors internally, depending upon the set value of `errno`, and return specific `HTTPD_SOCKET_ERR_` codes, which will eventually be conveyed as return value of `httpd_send()` function

Return

- Bytes : The number of bytes sent successfully
- `HTTPD_SOCKET_ERR_INVALID` : Invalid arguments
- `HTTPD_SOCKET_ERR_TIMEOUT` : Timeout/interrupted while calling socket `send()`
- `HTTPD_SOCKET_ERR_FAIL` : Unrecoverable error while calling socket `send()`

Parameters

- [in] `hd`: server instance
- [in] `sockfd`: session socket file descriptor
- [in] `buf`: buffer with bytes to send
- [in] `buf_len`: data size
- [in] `flags`: flags for the `send()` function

typedef int (*httpd_recv_func_t) (*httpd_handle_t* hd, int sockfd, char *buf, size_t buf_len, int flags)

Prototype for HTTPDs low-level recv function.

Note User specified recv function must handle errors internally, depending upon the set value of `errno`, and return specific `HTTPD_SOCKET_ERR_` codes, which will eventually be conveyed as return value of `httpd_req_recv()` function

Return

- Bytes : The number of bytes received successfully
- 0 : Buffer length parameter is zero / connection closed by peer
- `HTTPD_SOCKET_ERR_INVALID` : Invalid arguments
- `HTTPD_SOCKET_ERR_TIMEOUT` : Timeout/interrupted while calling socket `recv()`
- `HTTPD_SOCKET_ERR_FAIL` : Unrecoverable error while calling socket `recv()`

Parameters

- [in] `hd`: server instance
- [in] `sockfd`: session socket file descriptor
- [in] `buf`: buffer with bytes to send
- [in] `buf_len`: data size
- [in] `flags`: flags for the `send()` function

typedef int (*httpd_pending_func_t) (*httpd_handle_t* hd, int sockfd)

Prototype for HTTPDs low-level “get pending bytes” function.

Note User specified pending function must handle errors internally, depending upon the set value of `errno`, and return specific `HTTPD_SOCKET_ERR_` codes, which will be handled accordingly in the server task.

Return

- Bytes : The number of bytes waiting to be received
- `HTTPD_SOCKET_ERR_INVALID` : Invalid arguments
- `HTTPD_SOCKET_ERR_TIMEOUT` : Timeout/interrupted while calling socket `pending()`
- `HTTPD_SOCKET_ERR_FAIL` : Unrecoverable error while calling socket `pending()`

Parameters

- [in] `hd`: server instance
- [in] `sockfd`: session socket file descriptor

typedef *esp_err_t* (*httpd_err_handler_func_t) (*httpd_req_t* *req, *httpd_err_code_t* error)

Function prototype for HTTP error handling.

This function is executed upon HTTP errors generated during internal processing of an HTTP request. This is used to override the default behavior on error, which is to send HTTP error response and close the underlying socket.

Note

- If implemented, the server will not automatically send out HTTP error response codes, therefore, `httpd_resp_send_err()` must be invoked inside this function if user wishes to generate HTTP error responses.
- When invoked, the validity of `uri`, `method`, `content_len` and `user_ctx` fields of the `httpd_req_t` parameter is not guaranteed as the HTTP request may be partially received/parsed.
- The function must return `ESP_OK` if underlying socket needs to be kept open. Any other value will ensure that the socket is closed. The return value is ignored when error is of type `HTTPD_500_INTERNAL_SERVER_ERROR` and the socket closed anyway.

Return

- `ESP_OK` : error handled successful
- `ESP_FAIL` : failure indicates that the underlying socket needs to be closed

Parameters

- `[in]` `req`: HTTP request for which the error needs to be handled
- `[in]` `error`: Error type

typedef void **httpd_handle_t*
HTTP Server Instance Handle.

Every instance of the server will have a unique handle.

typedef enum http_method *httpd_method_t*
HTTP Method Type wrapper over “enum http_method” available in “http_parser” library.

typedef void (**httpd_free_ctx_fn_t*) (void *ctx)
Prototype for freeing context data (if any)

Parameters

- `[in]` `ctx`: object to free

typedef *esp_err_t* (**httpd_open_func_t*) (*httpd_handle_t* hd, int sockfd)
Function prototype for opening a session.

Called immediately after the socket was opened to set up the send/rcv functions and other parameters of the socket.

Return

- `ESP_OK` : On success
- Any value other than `ESP_OK` will signal the server to close the socket immediately

Parameters

- `[in]` `hd`: server instance
- `[in]` `sockfd`: session socket file descriptor

typedef void (**httpd_close_func_t*) (*httpd_handle_t* hd, int sockfd)
Function prototype for closing a session.

Note It’s possible that the socket descriptor is invalid at this point, the function is called for all terminated sessions. Ensure proper handling of return codes.

Parameters

- `[in]` `hd`: server instance
- `[in]` `sockfd`: session socket file descriptor

typedef bool (**httpd_uri_match_func_t*) (**const** char *reference_uri, **const** char *uri_to_match, size_t match_upto)

Function prototype for URI matching.

Return true on match

Parameters

- `[in]` `reference_uri`: URI/template with respect to which the other URI is matched
- `[in]` `uri_to_match`: URI/template being matched to the reference URI/template

- [in] `match_upto`: For specifying the actual length of `uri_to_match` up to which the matching algorithm is to be applied (The maximum value is `strlen(uri_to_match)`, independent of the length of `reference_uri`)

typedef struct *httpd_config* httpd_config_t

HTTP Server Configuration Structure.

Note Use `HTTPD_DEFAULT_CONFIG()` to initialize the configuration to a default value and then modify only those fields that are specifically determined by the use case.

typedef void (*httpd_work_fn_t) (void *arg)

Prototype of the HTTPD work function Please refer to `httpd_queue_work()` for more details.

Parameters

- [in] `arg`: The arguments for this work function

Enumerations

enum httpd_err_code_t

Error codes sent as HTTP response in case of errors encountered during processing of an HTTP request.

Values:

`HTTPD_500_INTERNAL_SERVER_ERROR = 0`

`HTTPD_501_METHOD_NOT_IMPLEMENTED`

`HTTPD_505_VERSION_NOT_SUPPORTED`

`HTTPD_400_BAD_REQUEST`

`HTTPD_404_NOT_FOUND`

`HTTPD_405_METHOD_NOT_ALLOWED`

`HTTPD_408_REQ_TIMEOUT`

`HTTPD_411_LENGTH_REQUIRED`

`HTTPD_414_URI_TOO_LONG`

`HTTPD_431_REQ_HDR_FIELDS_TOO_LARGE`

`HTTPD_ERR_CODE_MAX`

2.4.6 HTTPS server

Overview

This component is built on top of `esp_http_server`. The HTTPS server takes advantage of hooks and function overrides in the regular HTTP server to provide encryption using OpenSSL.

All documentation for `esp_http_server` applies also to a server you create this way.

Used APIs

The following API of `esp_http_server` should not be used with `esp_https_server`, as they are used internally to handle secure sessions and to maintain internal state:

- “send” , “receive” and “pending” function overrides - secure socket handling
 - `httpd_sess_set_send_override()`
 - `httpd_sess_set_rcv_override()`
 - `httpd_sess_set_pending_override()`
- “transport context” - both global and session
 - `httpd_sess_get_transport_ctx()` - returns SSL used for the session
 - `httpd_sess_set_transport_ctx()`

- `httpd_get_global_transport_ctx()` - returns the shared SSL context
- `httpd_config_t.global_transport_ctx`
- `httpd_config_t.global_transport_ctx_free_fn`
- `httpd_config_t.open_fn` - used to set up secure sockets

Everything else can be used without limitations.

Usage

Please see the example [protocols/https_server](#) to learn how to set up a secure server.

Basically all you need is to generate a certificate, embed it in the firmware, and provide its pointers and lengths to the start function via the init struct.

The server can be started with or without SSL by changing a flag in the init struct - `httpd_ssl_config.transport_mode`. This could be used e.g. for testing or in trusted environments where you prefer speed over security.

Performance

The initial session setup can take about two seconds, or more with slower clock speeds or more verbose logging. Subsequent requests through the open secure socket are much faster (down to under 100 ms).

API Reference

Header File

- [esp_https_server/include/esp_https_server.h](#)

Functions

`esp_err_t httpd_ssl_start` (*httpd_handle_t* *handle, *httpd_ssl_config_t* *config)

Create a SSL capable HTTP server (secure mode may be disabled in config)

Return success

Parameters

- [inout] config: - server config, must not be const. Does not have to stay valid after calling this function.
- [out] handle: - storage for the server handle, must be a valid pointer

void `httpd_ssl_stop` (*httpd_handle_t* handle)

Stop the server. Blocks until the server is shut down.

Parameters

- [in] handle:

Structures

`struct httpd_ssl_config`

HTTPS server config struct

Please use `HTTPD_SSL_CONFIG_DEFAULT()` to initialize it.

Public Members

httpd_config_t `httpd`

Underlying HTTPD server config

Parameters like task stack size and priority can be adjusted here.

```
const uint8_t *cacert_pem  
    CA certificate  
size_t cacert_len  
    CA certificate byte length  
const uint8_t *prvtkey_pem  
    Private key  
size_t prvtkey_len  
    Private key byte length  
httpd_ssl_transport_mode_t transport_mode  
    Transport Mode (default secure)  
uint16_t port_secure  
    Port used when transport mode is secure (default 443)  
uint16_t port_insecure  
    Port used when transport mode is insecure (default 80)
```

Macros

```
HTTPD_SSL_CONFIG_DEFAULT ()  
    Default config struct init  
  
(http_server default config had to be copied for customization)
```

Notes:

- port is set when starting the server, according to ‘transport_mode’
- one socket uses ~ 40kB RAM with SSL, we reduce the default socket count to 4
- SSL sockets are usually long-lived, closing LRU prevents pool exhaustion DOS
- Stack size may need adjustments depending on the user application

Type Definitions

```
typedef struct httpd_ssl_config httpd_ssl_config_t
```

Enumerations

```
enum httpd_ssl_transport_mode_t  
    Values:  
  
    HTTPD_SSL_TRANSPORT_SECURE  
  
    HTTPD_SSL_TRANSPORT_INSECURE
```

2.4.7 ICMP Echo

Overview

ICMP (Internet Control Message Protocol) is used for diagnostic or control purposes or generated in response to errors in IP operations. The common network util `ping` is implemented based on the ICMP packets with the type field value of 0, also called `Echo Reply`.

During a ping session, the source host firstly sends out an ICMP echo request packet and wait for an ICMP echo reply with specific times. In this way, it also measures the round-trip time for the messages. After receiving a valid ICMP echo reply, the source host will generate statistics about the IP link layer (e.g. packet loss, elapsed time, etc).

It is common that IoT device needs to check whether a remote server is alive or not. The device should show the warnings to users when it got offline. It can be achieved by creating a ping session and sending/parsing ICMP echo packets periodically.

To make this internal procedure much easier for users, ESP-IDF provides some out-of-box APIs.

Create a new ping session To create a ping session, you need to fill in the `esp_ping_config_t` configuration structure firstly, specifying target IP address, interval times, and etc. Optionally, you can also register some callback functions with the `esp_ping_callbacks_t` structure.

Example method to create a new ping session and register callbacks:

```
static void test_on_ping_success(esp_ping_handle_t hdl, void *args)
{
    // optionally, get callback arguments
    // const char* str = (const char*) args;
    // printf("%s\r\n", str); // "foo"
    uint8_t ttl;
    uint16_t seqno;
    uint32_t elapsed_time, recv_len;
    ip_addr_t target_addr;
    esp_ping_get_profile(hdl, ESP_PING_PROF_SEQNO, &seqno, sizeof(seqno));
    esp_ping_get_profile(hdl, ESP_PING_PROF_TTL, &ttl, sizeof(ttl));
    esp_ping_get_profile(hdl, ESP_PING_PROF_IPADDR, &target_addr, sizeof(target_
↪addr));
    esp_ping_get_profile(hdl, ESP_PING_PROF_SIZE, &recv_len, sizeof(recv_len));
    esp_ping_get_profile(hdl, ESP_PING_PROF_TIMEGAP, &elapsed_time, sizeof(elapsed_
↪time));
    printf("%d bytes from %s icmp_seq=%d ttl=%d time=%d ms\n",
           recv_len, inet_ntoa(target_addr.u_addr.ip4), seqno, ttl, elapsed_time);
}

static void test_on_ping_timeout(esp_ping_handle_t hdl, void *args)
{
    uint16_t seqno;
    ip_addr_t target_addr;
    esp_ping_get_profile(hdl, ESP_PING_PROF_SEQNO, &seqno, sizeof(seqno));
    esp_ping_get_profile(hdl, ESP_PING_PROF_IPADDR, &target_addr, sizeof(target_
↪addr));
    printf("From %s icmp_seq=%d timeout\n", inet_ntoa(target_addr.u_addr.ip4),
↪seqno);
}

static void test_on_ping_end(esp_ping_handle_t hdl, void *args)
{
    uint32_t transmitted;
    uint32_t received;
    uint32_t total_time_ms;

    esp_ping_get_profile(hdl, ESP_PING_PROF_REQUEST, &transmitted,
↪sizeof(transmitted));
    esp_ping_get_profile(hdl, ESP_PING_PROF_REPLY, &received, sizeof(received));
    esp_ping_get_profile(hdl, ESP_PING_PROF_DURATION, &total_time_ms, sizeof(total_
↪time_ms));
    printf("%d packets transmitted, %d received, time %dms\n", transmitted,
↪received, total_time_ms);
}

void initialize_ping()
{
    /* convert URL to IP address */
    ip_addr_t target_addr;
    struct addrinfo hint;
    struct addrinfo *res = NULL;
    memset(&hint, 0, sizeof(hint));
    memset(&target_addr, 0, sizeof(target_addr));
    getaddrinfo("www.espressif.com", NULL, &hint, &res);
    struct in_addr addr4 = ((struct sockaddr_in *) (res->ai_addr))->sin_addr;
    inet_addr_to_ip4addr(ip_2_ip4(&target_addr), &addr4);
}
```

(continues on next page)

(continued from previous page)

```

freeaddrinfo(res);

esp_ping_config_t ping_config = ESP_PING_DEFAULT_CONFIG();
ping_config.target_addr = target_addr;           // target IP address
ping_config.count = ESP_PING_COUNT_INFINITE;    // ping in infinite mode, esp_
↳ping_stop can stop it

/* set callback functions */
esp_ping_callbacks_t cbs;
cbs.on_ping_success = test_on_ping_success;
cbs.on_ping_timeout = test_on_ping_timeout;
cbs.on_ping_end = test_on_ping_end;
cbs.cb_args = "foo"; // arguments that will feed to all callback functions,
↳can be NULL
cbs.cb_args = eth_event_group;

esp_ping_handle_t ping;
esp_ping_new_session(&ping_config, &cbs, &ping);
}

```

Start and Stop ping session You can start and stop ping session with the handle returned by `esp_ping_new_session`. Note that, the ping session won't start automatically after creation. If the ping session is stopped, and restart again, the sequence number in ICMP packets will recount from zero again.

Delete a ping session If a ping session won't be used any more, you can delete it with `esp_ping_delete_session`. Please make sure the ping session is in stop state (i.e. you have called `esp_ping_stop` before or the ping session has finished all the procedures) when you call this function.

Get runtime statistics As the example code above, you can call `esp_ping_get_profile` to get different runtime statistics of ping session in the callback function.

Application Example

ICMP echo example: [protocols/icmp_echo](#)

API Reference

Header File

- `lwip/include/apps/ping/ping_sock.h`

Functions

`esp_err_t esp_ping_new_session(const esp_ping_config_t *config, const esp_ping_callbacks_t *cbs, esp_ping_handle_t *hdl_out)`

Create a ping session.

Return

- `ESP_ERR_INVALID_ARG`: invalid parameters (e.g. configuration is null, etc)
- `ESP_ERR_NO_MEM`: out of memory
- `ESP_FAIL`: other internal error (e.g. socket error)
- `ESP_OK`: create ping session successfully, user can take the ping handle to do follow-on jobs

Parameters

- `config`: ping configuration
- `cbs`: a bunch of callback functions invoked by internal ping task
- `hdl_out`: handle of ping session

esp_err_t **esp_ping_delete_session** (*esp_ping_handle_t* hdl)

Delete a ping session.

Return

- ESP_ERR_INVALID_ARG: invalid parameters (e.g. ping handle is null, etc)
- ESP_OK: delete ping session successfully

Parameters

- hdl: handle of ping session

esp_err_t **esp_ping_start** (*esp_ping_handle_t* hdl)

Start the ping session.

Return

- ESP_ERR_INVALID_ARG: invalid parameters (e.g. ping handle is null, etc)
- ESP_OK: start ping session successfully

Parameters

- hdl: handle of ping session

esp_err_t **esp_ping_stop** (*esp_ping_handle_t* hdl)

Stop the ping session.

Return

- ESP_ERR_INVALID_ARG: invalid parameters (e.g. ping handle is null, etc)
- ESP_OK: stop ping session successfully

Parameters

- hdl: handle of ping session

esp_err_t **esp_ping_get_profile** (*esp_ping_handle_t* hdl, *esp_ping_profile_t* profile, void *data, uint32_t size)

Get runtime profile of ping session.

Return

- ESP_ERR_INVALID_ARG: invalid parameters (e.g. ping handle is null, etc)
- ESP_ERR_INVALID_SIZE: the actual profile data size doesn't match the "size" parameter
- ESP_OK: get profile successfully

Parameters

- hdl: handle of ping session
- profile: type of profile
- data: profile data
- size: profile data size

Structures

struct esp_ping_callbacks_t

Type of "ping" callback functions.

Public Members

void ***cb_args**

arguments for callback functions

void (***on_ping_success**) (*esp_ping_handle_t* hdl, void *args)

Invoked by internal ping thread when received ICMP echo reply packet.

void (***on_ping_timeout**) (*esp_ping_handle_t* hdl, void *args)

Invoked by internal ping thread when receive ICMP echo reply packet timeout.

void (***on_ping_end**) (*esp_ping_handle_t* hdl, void *args)

Invoked by internal ping thread when a ping session is finished.

struct esp_ping_config_t

Type of "ping" configuration.

Public Members

- uint32_t count**
A “ping” session contains count procedures
- uint32_t interval_ms**
Milliseconds between each ping procedure
- uint32_t timeout_ms**
Timeout value (in milliseconds) of each ping procedure
- uint32_t data_size**
Size of the data next to ICMP packet header
- uint8_t tos**
Type of Service, a field specified in the IP header
- ip_addr_t target_addr**
Target IP address, either IPv4 or IPv6
- uint32_t task_stack_size**
Stack size of internal ping task
- uint32_t task_prio**
Priority of internal ping task

Macros

- ESP_PING_DEFAULT_CONFIG()**
Default ping configuration.
- ESP_PING_COUNT_INFINITE**
Set ping count to zero will ping target infinitely

Type Definitions

- typedef void *esp_ping_handle_t**
Type of “ping” session handle.

Enumerations

- enum esp_ping_profile_t**
Profile of ping session.
- Values:*
- ESP_PING_PROF_SEQNO**
Sequence number of a ping procedure
- ESP_PING_PROF_TTL**
Time to live of a ping procedure
- ESP_PING_PROF_REQUEST**
Number of request packets sent out
- ESP_PING_PROF_REPLY**
Number of reply packets received
- ESP_PING_PROF_IPADDR**
IP address of replied target
- ESP_PING_PROF_SIZE**
Size of received packet
- ESP_PING_PROF_TIMEGAP**
Elapsed time between request and reply packet

ESP_PING_PROF_DURATION

Elapsed time of the whole ping session

2.4.8 ESP Local Control

Overview

ESP Local Control (**esp_local_ctrl**) component in ESP-IDF provides capability to control an ESP device over Wi-Fi + HTTPS or BLE. It provides access to application defined **properties** that are available for reading / writing via a set of configurable handlers.

Initialization of the **esp_local_ctrl** service over BLE transport is performed as follows:

```
esp_local_ctrl_config_t config = {
    .transport = ESP_LOCAL_CTRL_TRANSPORT_BLE,
    .transport_config = {
        .ble = & (protocomm_ble_config_t) {
            .device_name = SERVICE_NAME,
            .service_uuid = {
                /* LSB <----->
                * -----> MSB */
                0x21, 0xd5, 0x3b, 0x8d, 0xbd, 0x75, 0x68, 0x8a,
                0xb4, 0x42, 0xeb, 0x31, 0x4a, 0x1e, 0x98, 0x3d
            }
        }
    },
    .handlers = {
        /* User defined handler functions */
        .get_prop_values = get_property_values,
        .set_prop_values = set_property_values,
        .usr_ctx = NULL,
        .usr_ctx_free_fn = NULL
    },
    /* Maximum number of properties that may be set */
    .max_properties = 10
};

/* Start esp_local_ctrl service */
ESP_ERROR_CHECK(esp_local_ctrl_start(&config));
```

Similarly for HTTPS transport:

```
/* Set the configuration */
httpd_ssl_config_t https_conf = HTTPD_SSL_CONFIG_DEFAULT();

/* Load server certificate */
extern const unsigned char cacert_pem_start[] asm("_binary_cacert_pem_
↪start");
extern const unsigned char cacert_pem_end[] asm("_binary_cacert_pem_end
↪");
https_conf.cacert_pem = cacert_pem_start;
https_conf.cacert_len = cacert_pem_end - cacert_pem_start;

/* Load server private key */
extern const unsigned char prvkey_pem_start[] asm("_binary_prvkey_pem_
↪start");
extern const unsigned char prvkey_pem_end[] asm("_binary_prvkey_pem_
↪end");
https_conf.prvkey_pem = prvkey_pem_start;
https_conf.prvkey_len = prvkey_pem_end - prvkey_pem_start;
```

(continues on next page)

(continued from previous page)

```

esp_local_ctrl_config_t config = {
    .transport = ESP_LOCAL_CTRL_TRANSPORT_HTTPD,
    .transport_config = {
        .httpd = &https_conf
    },
    .handlers = {
        /* User defined handler functions */
        .get_prop_values = get_property_values,
        .set_prop_values = set_property_values,
        .usr_ctx         = NULL,
        .usr_ctx_free_fn = NULL
    },
    /* Maximum number of properties that may be set */
    .max_properties = 10
};

/* Start esp_local_ctrl service */
ESP_ERROR_CHECK(esp_local_ctrl_start(&config));

```

Creating a property

Now that we know how to start the **esp_local_ctrl** service, let's add a property to it. Each property must have a unique *name* (string), a *type* (e.g. enum), *flags* (bit fields) and *size*.

The *size* is to be kept 0, if we want our property value to be of variable length (e.g. if its a string or bytestream). For fixed length property value data-types, like int, float, etc., setting the *size* field to the right value, helps **esp_local_ctrl** to perform internal checks on arguments received with write requests.

The interpretation of *type* and *flags* fields is totally upto the application, hence they may be used as enumerations, bit-fields, or even simple integers. One way is to use *type* values to classify properties, while *flags* to specify characteristics of a property.

Here is an example property which is to function as a timestamp. It is assumed that the application defines *TYPE_TIMESTAMP* and *READONLY*, which are used for setting the *type* and *flags* fields here.

```

/* Create a timestamp property */
esp_local_ctrl_prop_t timestamp = {
    .name       = "timestamp",
    .type       = TYPE_TIMESTAMP,
    .size       = sizeof(int32_t),
    .flags      = READONLY,
    .ctx        = func_get_time,
    .ctx_free_fn = NULL
};

/* Now register the property */
esp_local_ctrl_add_property(&timestamp);

```

Also notice that there is a *ctx* field, which is set to point to some custom *func_get_time()*. This can be used inside the property get / set handlers to retrieve timestamp.

Here is an example of *get_prop_values()* handler, which is used for retrieving the timestamp.

```

static esp_err_t get_property_values(size_t props_count,
                                     const esp_local_ctrl_prop_t *props,
                                     esp_local_ctrl_prop_val_t *prop_
→values,
                                     void *usr_ctx)
{
    for (uint32_t i = 0; i < props_count; i++) {

```

(continues on next page)

(continued from previous page)

```

ESP_LOGI(TAG, "Reading %s", props[i].name);
if (props[i].type == TYPE_TIMESTAMP) {
    /* Obtain the timer function from ctx */
    int32_t (*func_get_time)(void) = props[i].ctx;

    /* Use static variable for saving the value.
     * This is essential because the value has to be
     * valid even after this function returns.
     * Alternative is to use dynamic allocation
     * and set the free_fn field */
    static int32_t ts = func_get_time();
    prop_values[i].data = &ts;
}
}
return ESP_OK;
}

```

Here is an example of `set_prop_values()` handler. Notice how we restrict from writing to read-only properties.

```

static esp_err_t set_property_values(size_t props_count,
                                    const esp_local_ctrl_prop_t *props,
                                    const esp_local_ctrl_prop_val_t_
↳*prop_values,
                                    void *usr_ctx)
{
    for (uint32_t i = 0; i < props_count; i++) {
        if (props[i].flags & READONLY) {
            ESP_LOGE(TAG, "Cannot write to read-only property %s",
↳props[i].name);
            return ESP_ERR_INVALID_ARG;
        } else {
            ESP_LOGI(TAG, "Setting %s", props[i].name);

            /* For keeping it simple, lets only log the incoming data */
            ESP_LOG_BUFFER_HEX_LEVEL(TAG, prop_values[i].data,
                                     prop_values[i].size, ESP_LOG_INFO);
        }
    }
    return ESP_OK;
}

```

For complete example see [protocols/esp_local_ctrl](#)

Client Side Implementation

The client side implementation will have establish a protocomm session with the device first, over the supported mode of transport, and then send and receive protobuf messages understood by the `esp_local_ctrl` service. The service will translate these messages into requests and then call the appropriate handlers (set / get). Then, the generated response for each handler is again packed into a protobuf message and transmitted back to the client.

See below the various protobuf messages understood by the `esp_local_ctrl` service:

1. `get_prop_count` : This should simply return the total number of properties supported by the service
2. `get_prop_values` : This accepts an array of indices and should return the information (name, type, flags) and values of the properties corresponding to those indices
3. `set_prop_values` : This accepts an array of indices and an array of new values, which are used for setting the values of the properties corresponding to the indices

Note that indices may or may not be the same for a property, across multiple sessions. Therefore, the client must only use the names of the properties to uniquely identify them. So, every time a new session is established, the client should first call `get_prop_count` and then `get_prop_values`, hence form an index to name mapping for all properties.

Now when calling `set_prop_values` for a set of properties, it must first convert the names to indexes, using the created mapping. As emphasized earlier, the client must refresh the index to name mapping every time a new session is established with the same device.

The various protocomm endpoints provided by `esp_local_ctrl` are listed below:

Table 4: Endpoints provided by ESP Local Control

| Endpoint Name (BLE + GATT Server) | URI (HTTPS Server + mDNS) | Description |
|--------------------------------------|---|--|
| <code>esp_local_ctrl_version</code> | <code>https://<mdns-hostname>.local/esp_local_ctrl/version</code> | Endpoint used for retrieving version string |
| <code>esp_local_ctrl_control</code> | <code>https://<mdns-hostname>.local/esp_local_ctrl/control</code> | Endpoint used for sending / receiving control messages |

API Reference

Header File

- [esp_local_ctrl/include/esp_local_ctrl.h](#)

Functions

const `esp_local_ctrl_transport_t` ***esp_local_ctrl_get_transport_ble** (void)
Function for obtaining BLE transport mode.

const `esp_local_ctrl_transport_t` ***esp_local_ctrl_get_transport_httpd** (void)
Function for obtaining HTTPD transport mode.

`esp_err_t` **esp_local_ctrl_start** (**const** `esp_local_ctrl_config_t` **config*)
Start local control service.

Return

- ESP_OK : Success
- ESP_FAIL : Failure

Parameters

- [in] *config*: Pointer to configuration structure

`esp_err_t` **esp_local_ctrl_stop** (void)
Stop local control service.

`esp_err_t` **esp_local_ctrl_add_property** (**const** `esp_local_ctrl_prop_t` **prop*)
Add a new property.

This adds a new property and allocates internal resources for it. The total number of properties that could be added is limited by configuration option `max_properties`

Return

- ESP_OK : Success
- ESP_FAIL : Failure

Parameters

- [in] *prop*: Property description structure

`esp_err_t` **esp_local_ctrl_remove_property** (**const** char **name*)
Remove a property.

This finds a property by name, and releases the internal resources which are associated with it.

Return

- ESP_OK : Success
- ESP_ERR_NOT_FOUND : Failure

Parameters

- [in] name: Name of the property to remove

const *esp_local_ctrl_prop_t* ***esp_local_ctrl_get_property** (**const** char *name)

Get property description structure by name.

This API may be used to get a property's context structure *esp_local_ctrl_prop_t* when its name is known

Return

- Pointer to property
- NULL if not found

Parameters

- [in] name: Name of the property to find

esp_err_t **esp_local_ctrl_set_handler** (**const** char *ep_name, *protocomm_req_handler_t* handler, void *user_ctx)

Register protocomm handler for a custom endpoint.

This API can be called by the application to register a protocomm handler for an endpoint after the local control service has started.

Note In case of BLE transport the names and uuids of all custom endpoints must be provided beforehand as a part of the *protocomm_ble_config_t* structure set in *esp_local_ctrl_config_t*, and passed to *esp_local_ctrl_start()*.

Return

- ESP_OK : Success
- ESP_FAIL : Failure

Parameters

- [in] ep_name: Name of the endpoint
- [in] handler: Endpoint handler function
- [in] user_ctx: User data

Unions

union *esp_local_ctrl_transport_config_t*

#include <esp_local_ctrl.h> Transport mode (BLE / HTTPD) configuration.

Public Members

esp_local_ctrl_transport_config_ble_t *ble

This is same as *protocomm_ble_config_t*. See *protocomm_ble.h* for available configuration parameters.

esp_local_ctrl_transport_config_httpd_t *httpd

This is same as *httpd_ssl_config_t*. See *esp_https_server.h* for available configuration parameters.

Structures

struct *esp_local_ctrl_prop*

Property description data structure, which is to be populated and passed to the *esp_local_ctrl_add_property()* function.

Once a property is added, its structure is available for read-only access inside *get_prop_values()* and *set_prop_values()* handlers.

Public Members

char *name

Unique name of property

uint32_t type

Type of property. This may be set to application defined enums

size_t size

Size of the property value, which:

- if zero, the property can have values of variable size
- if non-zero, the property can have values of fixed size only, therefore, checks are performed internally by `esp_local_ctrl` when setting the value of such a property

uint32_t flags

Flags set for this property. This could be a bit field. A flag may indicate property behavior, e.g. read-only / constant

void *ctx

Pointer to some context data relevant for this property. This will be available for use inside the `get_prop_values` and `set_prop_values` handlers as a part of this property structure. When set, this is valid throughout the lifetime of a property, till either the property is removed or the `esp_local_ctrl` service is stopped.

void (*ctx_free_fn) (void *ctx)

Function used by `esp_local_ctrl` to internally free the property context when `esp_local_ctrl_remove_property()` or `esp_local_ctrl_stop()` is called.

struct esp_local_ctrl_prop_val

Property value data structure. This gets passed to the `get_prop_values()` and `set_prop_values()` handlers for the purpose of retrieving or setting the present value of a property.

Public Members**void *data**

Pointer to memory holding property value

size_t size

Size of property value

void (*free_fn) (void *data)

This may be set by the application in `get_prop_values()` handler to tell `esp_local_ctrl` to call this function on the data pointer above, for freeing its resources after sending the `get_prop_values` response.

struct esp_local_ctrl_handlers

Handlers for receiving and responding to local control commands for getting and setting properties.

Public Members

esp_err_t (*get_prop_values) (size_t props_count, const esp_local_ctrl_prop_t props[], esp_local_ctrl_prop_val_t prop_values[], void *usr_ctx)

Handler function to be implemented for retrieving current values of properties.

Note If any of the properties have fixed sizes, the size field of corresponding element in `prop_values` need to be set

Return Returning different error codes will convey the corresponding protocol level errors to the client

:

- `ESP_OK` : Success
- `ESP_ERR_INVALID_ARG` : InvalidArgument
- `ESP_ERR_INVALID_STATE` : InvalidProto
- All other error codes : InternalError

Parameters

- [in] `props_count`: Total elements in the `props` array
- [in] `props`: Array of properties, the current values for which have been requested by the client

- [out] `prop_values`: Array of empty property values, the elements of which need to be populated with the current values of those properties specified by `props` argument
- [in] `usr_ctx`: This provides value of the `usr_ctx` field of `esp_local_ctrl_handlers_t` structure

`esp_err_t` (***set_prop_values**) (size_t `props_count`, const `esp_local_ctrl_prop_t` `props`[], const `esp_local_ctrl_prop_val_t` `prop_values`[], void *`usr_ctx`)

Handler function to be implemented for changing values of properties.

Note If any of the properties have variable sizes, the size field of the corresponding element in `prop_values` must be checked explicitly before making any assumptions on the size.

Return Returning different error codes will convey the corresponding protocol level errors to the client :

- `ESP_OK` : Success
- `ESP_ERR_INVALID_ARG` : InvalidArgument
- `ESP_ERR_INVALID_STATE` : InvalidProto
- All other error codes : InternalError

Parameters

- [in] `props_count`: Total elements in the `props` array
- [in] `props`: Array of properties, the values for which the client requests to change
- [in] `prop_values`: Array of property values, the elements of which need to be used for updating those properties specified by `props` argument
- [in] `usr_ctx`: This provides value of the `usr_ctx` field of `esp_local_ctrl_handlers_t` structure

void ***usr_ctx**

Context pointer to be passed to above handler functions upon invocation. This is different from the property level context, as this is valid throughout the lifetime of the `esp_local_ctrl` service, and freed only when the service is stopped.

void (***usr_ctx_free_fn**) (void *`usr_ctx`)

Pointer to function which will be internally invoked on `usr_ctx` for freeing the context resources when `esp_local_ctrl_stop()` is called.

struct esp_local_ctrl_config

Configuration structure to pass to `esp_local_ctrl_start()`

Public Members

const `esp_local_ctrl_transport_t` ***transport**

Transport layer over which service will be provided

`esp_local_ctrl_transport_config_t` **transport_config**

Transport layer over which service will be provided

`esp_local_ctrl_handlers_t` **handlers**

Register handlers for responding to get/set requests on properties

size_t **max_properties**

This limits the number of properties that are available at a time

Macros

`ESP_LOCAL_CTRL_TRANSPORT_BLE`

`ESP_LOCAL_CTRL_TRANSPORT_HTTPD`

Type Definitions

typedef struct esp_local_ctrl_prop esp_local_ctrl_prop_t

Property description data structure, which is to be populated and passed to the `esp_local_ctrl_add_property()` function.

Once a property is added, its structure is available for read-only access inside `get_prop_values()` and `set_prop_values()` handlers.

```
typedef struct esp_local_ctrl_prop_val esp_local_ctrl_prop_val_t
```

Property value data structure. This gets passed to the `get_prop_values()` and `set_prop_values()` handlers for the purpose of retrieving or setting the present value of a property.

```
typedef struct esp_local_ctrl_handlers esp_local_ctrl_handlers_t
```

Handlers for receiving and responding to local control commands for getting and setting properties.

```
typedef struct esp_local_ctrl_transport esp_local_ctrl_transport_t
```

Transport mode (BLE / HTTPD) over which the service will be provided.

This is forward declaration of a private structure, implemented internally by `esp_local_ctrl`.

```
typedef struct protocomm_ble_config esp_local_ctrl_transport_config_ble_t
```

Configuration for transport mode BLE.

This is a forward declaration for `protocomm_ble_config_t`. To use this, application must set `CONFIG_BT_BLUEDROID_ENABLED` and include `protocomm_ble.h`.

```
typedef struct httpd_ssl_config esp_local_ctrl_transport_config_httpd_t
```

Configuration for transport mode HTTPD.

This is a forward declaration for `httpd_ssl_config_t`. To use this, application must set `CONFIG_ESP_HTTPS_SERVER_ENABLE` and include `esp_https_server.h`.

```
typedef struct esp_local_ctrl_config esp_local_ctrl_config_t
```

Configuration structure to pass to `esp_local_ctrl_start()`

2.4.9 mDNS Service

Overview

mDNS is a multicast UDP service that is used to provide local network service and host discovery.

mDNS is installed by default on most operating systems or is available as separate package. On Mac OS it is installed by default and is called Bonjour. Apple releases an installer for Windows that can be found [on Apple's support page](#). On Linux, mDNS is provided by [avahi](#) and is usually installed by default.

mDNS Properties

- `hostname`: the hostname that the device will respond to. If not set, the hostname will be read from the interface. Example: `my-esp32` will resolve to `my-esp32.local`
- `default_instance`: friendly name for your device, like Jhon's ESP32 Thing. If not set, `hostname` will be used.

Example method to start mDNS for the STA interface and set `hostname` and `default_instance`:

```
void start_mdns_service()
{
    //initialize mDNS service
    esp_err_t err = mdns_init();
    if (err) {
        printf("MDNS Init failed: %d\n", err);
        return;
    }

    //set hostname
    mdns_hostname_set("my-esp32");
    //set default instance
    mdns_instance_name_set("Jhon's ESP32 Thing");
}
```

mDNS Services mDNS can advertise information about network services that your device offers. Each service is defined by a few properties.

- `instance_name`: friendly name for your service, like Jhon's ESP32 Web Server. If not defined, `default_instance` will be used.
- `service_type`: (required) service type, prepended with underscore. Some common types can be found [here](#).
- `proto`: (required) protocol that the service runs on, prepended with underscore. Example: `_tcp` or `_udp`
- `port`: (required) network port that the service runs on
- `txt`: {var, val} array of strings, used to define properties for your service

Example method to add a few services and different properties:

```
void add_mdns_services()
{
    //add our services
    mdns_service_add(NULL, "_http", "_tcp", 80, NULL, 0);
    mdns_service_add(NULL, "_arduino", "_tcp", 3232, NULL, 0);
    mdns_service_add(NULL, "_myservice", "_udp", 1234, NULL, 0);

    //NOTE: services must be added before their properties can be set
    //use custom instance for the web server
    mdns_service_instance_name_set("_http", "_tcp", "Jhon's ESP32 Web Server");

    mdns_txt_item_t serviceTxtData[3] = {
        {"board", "esp32"},
        {"u", "user"},
        {"p", "password"}
    };
    //set txt data for service (will free and replace current data)
    mdns_service_txt_set("_http", "_tcp", serviceTxtData, 3);

    //change service port
    mdns_service_port_set("_myservice", "_udp", 4321);
}
```

mDNS Query mDNS provides methods for browsing for services and resolving host's IP/IPv6 addresses.

Results for services are returned as a linked list of `mdns_result_t` objects.

Example method to resolve host IPs:

```
void resolve_mdns_host(const char * host_name)
{
    printf("Query A: %s.local", host_name);

    struct ip4_addr addr;
    addr.addr = 0;

    esp_err_t err = mdns_query_a(host_name, 2000, &addr);
    if(err){
        if(err == ESP_ERR_NOT_FOUND){
            printf("Host was not found!");
            return;
        }
        printf("Query Failed");
        return;
    }

    printf(IPSTR, IP2STR(&addr));
}
```

Example method to resolve local services:


```

static const char * if_str[] = {"STA", "AP", "ETH", "MAX"};
static const char * ip_protocol_str[] = {"V4", "V6", "MAX"};

void mdns_print_results(mdns_result_t * results){
    mdns_result_t * r = results;
    mdns_ip_addr_t * a = NULL;
    int i = 1, t;
    while(r){
        printf("%d: Interface: %s, Type: %s\n", i++, if_str[r->tcpip_if], ip_
↳protocol_str[r->ip_protocol]);
        if(r->instance_name){
            printf(" PTR : %s\n", r->instance_name);
        }
        if(r->hostname){
            printf(" SRV : %s.local:%u\n", r->hostname, r->port);
        }
        if(r->txt_count){
            printf(" TXT : [%u] ", r->txt_count);
            for(t=0; t<r->txt_count; t++){
                printf("%s=%s; ", r->txt[t].key, r->txt[t].value);
            }
            printf("\n");
        }
        a = r->addr;
        while(a){
            if(a->addr.type == IPADDR_TYPE_V6){
                printf(" AAAA: " IPV6STR "\n", IPV62STR(a->addr.u_addr.ip6));
            } else {
                printf(" A : " IPSTR "\n", IP2STR(&(a->addr.u_addr.ip4)));
            }
            a = a->next;
        }
        r = r->next;
    }
}

void find_mdns_service(const char * service_name, const char * proto)
{
    ESP_LOGI(TAG, "Query PTR: %s.%s.local", service_name, proto);

    mdns_result_t * results = NULL;
    esp_err_t err = mdns_query_ptr(service_name, proto, 3000, 20, &results);
    if(err){
        ESP_LOGE(TAG, "Query Failed");
        return;
    }
    if(!results){
        ESP_LOGW(TAG, "No results found!");
        return;
    }

    mdns_print_results(results);
    mdns_query_results_free(results);
}

```

Example of using the methods above:

```

void my_app_some_method(){
    //search for esp32-mdns.local
    resolve_mdns_host("esp32-mdns");
}

```

(continues on next page)

(continued from previous page)

```
//search for HTTP servers
find_mdns_service("_http", "_tcp");
//or file servers
find_mdns_service("_smb", "_tcp"); //windows sharing
find_mdns_service("_afpovertcp", "_tcp"); //apple sharing
find_mdns_service("_nfs", "_tcp"); //NFS server
find_mdns_service("_ftp", "_tcp"); //FTP server
//or networked printer
find_mdns_service("_printer", "_tcp");
find_mdns_service("_ipp", "_tcp");
}
```

Application Example

mDNS server/scanner example: [protocols/mdns](#).

API Reference

Header File

- [mdns/include/mdns.h](#)

Functions

esp_err_t **mdns_init** (void)

Initialize mDNS on given interface.

Return

- ESP_OK on success
- ESP_ERR_INVALID_STATE when failed to register event handler
- ESP_ERR_NO_MEM on memory error
- ESP_FAIL when failed to start mdns task

void **mdns_free** (void)

Stop and free mDNS server.

esp_err_t **mdns_hostname_set** (const char *hostname)

Set the hostname for mDNS server required if you want to advertise services.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_NO_MEM memory error

Parameters

- hostname: Hostname to set

esp_err_t **mdns_instance_name_set** (const char *instance_name)

Set the default instance name for mDNS server.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_NO_MEM memory error

Parameters

- instance_name: Instance name to set

esp_err_t **mdns_service_add** (const char *instance_name, const char *service_type, const char *proto, uint16_t port, *mdns_txt_item_t* txt[], size_t num_items)

Add service to mDNS server.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_NO_MEM memory error
- ESP_FAIL failed to add service

Parameters

- *instance_name*: instance name to set. If NULL, global instance name or hostname will be used
- *service_type*: service type (`_http`, `_ftp`, etc)
- *proto*: service protocol (`_tcp`, `_udp`)
- *port*: service port
- *txt*: string array of TXT data (eg. `{{ "var" , " val" },{ "other" , " 2" }}`)
- *num_items*: number of items in TXT data

esp_err_t **mdns_service_remove** (**const** char **service_type*, **const** char **proto*)

Remove service from mDNS server.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_NOT_FOUND Service not found
- ESP_ERR_NO_MEM memory error

Parameters

- *service_type*: service type (`_http`, `_ftp`, etc)
- *proto*: service protocol (`_tcp`, `_udp`)

esp_err_t **mdns_service_instance_name_set** (**const** char **service_type*, **const** char **proto*,
const char **instance_name*)

Set instance name for service.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_NOT_FOUND Service not found
- ESP_ERR_NO_MEM memory error

Parameters

- *service_type*: service type (`_http`, `_ftp`, etc)
- *proto*: service protocol (`_tcp`, `_udp`)
- *instance_name*: instance name to set

esp_err_t **mdns_service_port_set** (**const** char **service_type*, **const** char **proto*, `uint16_t` *port*)

Set service port.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_NOT_FOUND Service not found
- ESP_ERR_NO_MEM memory error

Parameters

- *service_type*: service type (`_http`, `_ftp`, etc)
- *proto*: service protocol (`_tcp`, `_udp`)
- *port*: service port

esp_err_t **mdns_service_txt_set** (**const** char **service_type*, **const** char **proto*, *mdns_txt_item_t*
txt[], `uint8_t` *num_items*)

Replace all TXT items for service.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_NOT_FOUND Service not found
- ESP_ERR_NO_MEM memory error

Parameters

- *service_type*: service type (`_http`, `_ftp`, etc)
- *proto*: service protocol (`_tcp`, `_udp`)

- `txt`: array of TXT data (eg. {{ “var” ,” val” },{ “other” ,” 2” }})
- `num_items`: number of items in TXT data

esp_err_t **mdns_service_txt_item_set** (**const** char **service_type*, **const** char **proto*, **const** char **key*, **const** char **value*)

Set/Add TXT item for service TXT record.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_NOT_FOUND Service not found
- ESP_ERR_NO_MEM memory error

Parameters

- *service_type*: service type (`_http`, `_ftp`, etc)
- *proto*: service protocol (`_tcp`, `_udp`)
- *key*: the key that you want to add/update
- *value*: the new value of the key

esp_err_t **mdns_service_txt_item_remove** (**const** char **service_type*, **const** char **proto*, **const** char **key*)

Remove TXT item for service TXT record.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_NOT_FOUND Service not found
- ESP_ERR_NO_MEM memory error

Parameters

- *service_type*: service type (`_http`, `_ftp`, etc)
- *proto*: service protocol (`_tcp`, `_udp`)
- *key*: the key that you want to remove

esp_err_t **mdns_service_remove_all** (void)

Remove and free all services from mDNS server.

Return

- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

esp_err_t **mdns_query** (**const** char **name*, **const** char **service_type*, **const** char **proto*, uint16_t *type*, uint32_t *timeout*, size_t *max_results*, *mdns_result_t* ***results*)

Query mDNS for host or service All following query methods are derived from this one.

Return

- ESP_OK success
- ESP_ERR_INVALID_STATE mDNS is not running
- ESP_ERR_NO_MEM memory error
- ESP_ERR_INVALID_ARG timeout was not given

Parameters

- *name*: service instance or host name (NULL for PTR queries)
- *service_type*: service type (`_http`, `_arduino`, `_ftp` etc.) (NULL for host queries)
- *proto*: service protocol (`_tcp`, `_udp`, etc.) (NULL for host queries)
- *type*: type of query (`MDNS_TYPE_*`)
- *timeout*: time in milliseconds to wait for answers.
- *max_results*: maximum results to be collected
- *results*: pointer to the results of the query results must be freed using `mdns_query_results_free` below

void **mdns_query_results_free** (*mdns_result_t* **results*)

Free query results.

Parameters

- *results*: linked list of results to be freed

esp_err_t **mdns_query_ptr**(**const** char **service_type*, **const** char **proto*, uint32_t *timeout*, size_t *max_results*, *mdns_result_t* ***results*)

Query mDNS for service.

Return

- ESP_OK success
- ESP_ERR_INVALID_STATE mDNS is not running
- ESP_ERR_NO_MEM memory error
- ESP_ERR_INVALID_ARG parameter error

Parameters

- *service_type*: service type (`_http`, `_arduino`, `_ftp` etc.)
- *proto*: service protocol (`_tcp`, `_udp`, etc.)
- *timeout*: time in milliseconds to wait for answer.
- *max_results*: maximum results to be collected
- *results*: pointer to the results of the query

esp_err_t **mdns_query_srv**(**const** char **instance_name*, **const** char **service_type*, **const** char **proto*, uint32_t *timeout*, *mdns_result_t* ***result*)

Query mDNS for SRV record.

Return

- ESP_OK success
- ESP_ERR_INVALID_STATE mDNS is not running
- ESP_ERR_NO_MEM memory error
- ESP_ERR_INVALID_ARG parameter error

Parameters

- *instance_name*: service instance name
- *service_type*: service type (`_http`, `_arduino`, `_ftp` etc.)
- *proto*: service protocol (`_tcp`, `_udp`, etc.)
- *timeout*: time in milliseconds to wait for answer.
- *result*: pointer to the result of the query

esp_err_t **mdns_query_txt**(**const** char **instance_name*, **const** char **service_type*, **const** char **proto*, uint32_t *timeout*, *mdns_result_t* ***result*)

Query mDNS for TXT record.

Return

- ESP_OK success
- ESP_ERR_INVALID_STATE mDNS is not running
- ESP_ERR_NO_MEM memory error
- ESP_ERR_INVALID_ARG parameter error

Parameters

- *instance_name*: service instance name
- *service_type*: service type (`_http`, `_arduino`, `_ftp` etc.)
- *proto*: service protocol (`_tcp`, `_udp`, etc.)
- *timeout*: time in milliseconds to wait for answer.
- *result*: pointer to the result of the query

esp_err_t **mdns_query_a**(**const** char **host_name*, uint32_t *timeout*, esp_ip4_addr_t **addr*)

Query mDNS for A record.

Return

- ESP_OK success
- ESP_ERR_INVALID_STATE mDNS is not running
- ESP_ERR_NO_MEM memory error
- ESP_ERR_INVALID_ARG parameter error

Parameters

- *host_name*: host name to look for
- *timeout*: time in milliseconds to wait for answer.
- *addr*: pointer to the resulting IP4 address

esp_err_t **mdns_query_aaaa**(**const** char **host_name*, uint32_t *timeout*, esp_ip6_addr_t **addr*)

Query mDNS for A record.

Return

- ESP_OK success
- ESP_ERR_INVALID_STATE mDNS is not running
- ESP_ERR_NO_MEM memory error
- ESP_ERR_INVALID_ARG parameter error

Parameters

- `host_name`: host name to look for
- `timeout`: time in milliseconds to wait for answer. If 0, `max_results` needs to be defined
- `addr`: pointer to the resulting IP6 address

esp_err_t **mdns_handle_system_event** (void *ctx, *system_event_t* *event)

System event handler This method controls the service state on all active interfaces and applications are required to call it from the system event handler for normal operation of mDNS service.

Parameters

- `ctx`: The system event context
- `event`: The system event

Structures

struct mdns_txt_item_t

mDNS basic text item structure Used in `mdns_service_add()`

Public Members

const char *key
item key name

const char *value
item value string

struct mdns_ip_addr_s

mDNS query linked list IP item

Public Members

esp_ip_addr_t addr
IP address

struct mdns_ip_addr_s *next
next IP, or NULL for the last IP in the list

struct mdns_result_s

mDNS query result structure

Public Members

struct mdns_result_s *next
next result, or NULL for the last result in the list

mdns_if_t tcpip_if
interface index

mdns_ip_protocol_t ip_protocol
ip_protocol type of the interface (v4/v6)

char *instance_name
instance name

char *hostname
hostname

`uint16_t port`
service port

`mdns_txt_item_t *txt`
txt record

`size_t txt_count`
number of txt items

`mdns_ip_addr_t *addr`
linked list of IP addresses found

Macros

`MDNS_TYPE_A`
`MDNS_TYPE_PTR`
`MDNS_TYPE_TXT`
`MDNS_TYPE_AAAA`
`MDNS_TYPE_SRV`
`MDNS_TYPE_OPT`
`MDNS_TYPE_NSEC`
`MDNS_TYPE_ANY`

Type Definitions

`typedef struct mdns_ip_addr_s mdns_ip_addr_t`
mDNS query linked list IP item

`typedef enum mdns_if_internal mdns_if_t`

`typedef struct mdns_result_s mdns_result_t`
mDNS query result structure

Enumerations

`enum mdns_ip_protocol_t`
mDNS enum to specify the ip_protocol type

Values:

`MDNS_IP_PROTOCOL_V4`
`MDNS_IP_PROTOCOL_V6`
`MDNS_IP_PROTOCOL_MAX`

`enum mdns_if_internal`

Values:

`MDNS_IF_STA = 0`
`MDNS_IF_AP = 1`
`MDNS_IF_ETH = 2`
`MDNS_IF_MAX`

2.4.10 ESP-Modbus

Overview

The Modbus serial communication protocol is de facto standard protocol widely used to connect industrial electronic devices. Modbus allows communication among many devices connected to the same network, for example, a system

that measures temperature and humidity and communicates the results to a computer. The Modbus protocol uses several types of data: Holding Registers, Input Registers, Coils (single bit output), Discrete Inputs. Versions of the Modbus protocol exist for serial port and for Ethernet and other protocols that support the Internet protocol suite. There are many variants of Modbus protocols, some of them are:

- `Modbus RTU`—This is used in serial communication and makes use of a compact, binary representation of the data for protocol communication. The RTU format follows the commands/data with a cyclic redundancy check checksum as an error check mechanism to ensure the reliability of data. Modbus RTU is the most common implementation available for Modbus. A Modbus RTU message must be transmitted continuously without inter-character hesitations. Modbus messages are framed (separated) by idle (silent) periods. The RS-485 interface communication is usually used for this type.
- `Modbus ASCII`—This is used in serial communication and makes use of ASCII characters for protocol communication. The ASCII format uses a longitudinal redundancy check checksum. Modbus ASCII messages are framed by leading colon (“:”) and trailing newline (CR/LF).
- `Modbus TCP/IP` or `Modbus TCP`—This is a Modbus variant used for communications over TCP/IP networks, connecting over port 502. It does not require a checksum calculation, as lower layers already provide checksum protection.

Modbus common interface API overview The API functions below provide common functionality to setup Modbus stack for slave and master implementation accordingly. ISP-IDF supports Modbus serial slave and master protocol stacks and provides `modbus_controller` interface API to interact with user application.

`esp_err_t mbc_slave_init` (`mb_port_type_t port_type`, `void **handler`)

Initialize Modbus controller and stack.

Return

- `ESP_OK` Success
- `ESP_ERR_NO_MEM` Parameter error

Parameters

- [out] `handler`: handler(pointer) to master data structure
- [in] `port_type`: type of stack

`esp_err_t mbc_master_init` (`mb_port_type_t port_type`, `void **handler`)

Initialize Modbus controller and stack.

Return

- `ESP_OK` Success
- `ESP_ERR_NO_MEM` Parameter error

Parameters

- [out] `handler`: handler(pointer) to master data structure
- [in] `port_type`: the type of port

The function initializes the Modbus controller interface and its active context (tasks, RTOS objects and other resources).

`esp_err_t mbc_slave_setup` (`void *comm_info`)

Set Modbus communication parameters for the controller.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Incorrect parameter data

Parameters

- `comm_info`: Communication parameters structure.

`esp_err_t mbc_master_setup` (`void *comm_info`)

Set Modbus communication parameters for the controller.

Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Incorrect parameter data

Parameters

- `comm_info`: Communication parameters structure.

The function is used to setup communication parameters of the Modbus stack. See the Modbus controller API documentation for more information.

mbc_slave_set_descriptor(): Initialization of slave descriptor.

mbc_master_set_descriptor(): Initialization of master descriptor.

The Modbus stack uses parameter description tables (descriptors) for communication. These are different for master and slave implementation of stack and should be assigned by the API call before start of communication.

esp_err_t **mbc_slave_start** (void)

Start Modbus communication stack.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Modbus stack start error

esp_err_t **mbc_master_start** (void)

Start Modbus communication stack.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Modbus stack start error

Modbus controller start function. Starts stack and interface and allows communication.

esp_err_t **mbc_slave_destroy** (void)

Destroy Modbus controller and stack.

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE Parameter error

esp_err_t **mbc_master_destroy** (void)

Destroy Modbus controller and stack.

Return

- ESP_OK Success
- ESP_ERR_INVALID_STATE Parameter error

This function stops Modbus communication stack and destroys controller interface.

There are some configurable parameters of modbus_controller interface and Modbus stack that can be configured using KConfig values in “Modbus configuration” menu. The most important option in KConfig menu is “Selection of Modbus stack support mode” that allows to select master or slave stack for implementation. See the examples for more information about how to use these API functions.

Modbus serial slave interface API overview The slave stack requires the user defined structures which represent Modbus parameters accessed by stack. These structures should be prepared by user and be assigned to the modbus_controller interface using *mbc_slave_set_descriptor()* API call before start of communication. The interface API functions below are used for Modbus slave application:

esp_err_t **mbc_slave_set_descriptor** (mb_register_area_descriptor_t *descr_data*)

Set Modbus area descriptor.

Return

- ESP_OK: The appropriate descriptor is set
- ESP_ERR_INVALID_ARG: The argument is incorrect

Parameters

- *descr_data*: Modbus registers area descriptor structure

The function initializes Modbus communication descriptors for each type of Modbus register area (Holding Registers, Input Registers, Coils (single bit output), Discrete Inputs). Once areas are initialized and the *mbc_slave_start()* API is called the Modbus stack can access the data in user data structures by request from master. See the *mb_register_area_descriptor_t* and example for more information.

`mb_event_group_t mbc_slave_check_event (mb_event_group_t group)`

Wait for specific event on parameter change.

Return

- `mb_event_group_t` event bits triggered

Parameters

- `group`: Group event bit mask to wait for change

The blocking call to function waits for event specified in the input parameter as event mask. Once master access the parameter and event mask matches the parameter the application task will be unblocked and function will return `ESP_OK`. See the `mb_event_group_t` for more information about Modbus event masks.

`esp_err_t mbc_slave_get_param_info (mb_param_info_t *reg_info, uint32_t timeout)`

Get parameter information.

Return

- `ESP_OK` Success
- `ESP_ERR_TIMEOUT` Can not get data from parameter queue or queue overflow

Parameters

- `[out] reg_info`: parameter info structure
- `timeout`: Timeout in milliseconds to read information from parameter queue

The function gets information about accessed parameters from modbus controller event queue. The KConfig ‘`CONFIG_FMB_CONTROLLER_NOTIFY_QUEUE_SIZE`’ key can be used to configure the notification queue size. The timeout parameter allows to specify timeout for waiting notification. The `mb_param_info_t` structure contain information about accessed parameter.

Modbus serial master interface API overview The Modbus master implementation requires parameter description table be defined before start of stack. This table describes characteristics (physical parameters like temperature, humidity, etc.) and links them to Modbus registers in specific slave device in the Modbus segment. The table has to be assigned to the `modbus_controller` interface using `mbc_master_set_descriptor()` API call before start of communication.

Below are the interface API functions that are used to setup and use Modbus master stack from user application and can be executed in next order:

`esp_err_t mbc_master_set_descriptor (const mb_parameter_descriptor_t *descriptor, const uint16_t num_elements)`

Assign parameter description table for Modbus controller interface.

Return

- `esp_err_t ESP_OK` - set descriptor successfully
- `esp_err_t ESP_ERR_INVALID_ARG` - invalid argument in function call

Parameters

- `[in] descriptor`: pointer to parameter description table
- `num_elements`: number of elements in the table

Assigns parameter description table for Modbus controller interface. The table has to be prepared by user according to particular

`esp_err_t mbc_master_send_request (mb_param_request_t *request, void *data_ptr)`

Send data request as defined in parameter request, waits response from slave and returns status of command execution. This function provides standard way for read/write access to Modbus devices in the network.

Return

- `esp_err_t ESP_OK` - request was successful
- `esp_err_t ESP_ERR_INVALID_ARG` - invalid argument of function
- `esp_err_t ESP_ERR_INVALID_RESPONSE` - an invalid response from slave
- `esp_err_t ESP_ERR_TIMEOUT` - operation timeout or no response from slave
- `esp_err_t ESP_ERR_NOT_SUPPORTED` - the request command is not supported by slave
- `esp_err_t ESP_FAIL` - slave returned an exception or other failure

Parameters

- `[in] request`: pointer to request structure of type `mb_param_request_t`

- [in] `data_ptr`: pointer to data buffer to send or received data (dependent of command field in request)

This function sends data request as defined in parameter request, waits response from corresponded slave and returns status of command execution. This function provides a standard way for read/write access to Modbus devices in the network.

```
esp_err_t mbc_master_get_cid_info(uint16_t cid, const mb_parameter_descriptor_t  
                                **param_info)
```

Get information about supported characteristic defined as cid. Uses parameter description table to get this information. The function will check if characteristic defined as a cid parameter is supported and returns its description in param_info. Returns ESP_ERR_NOT_FOUND if characteristic is not supported.

Return

- `esp_err_t` ESP_OK - request was successful and buffer contains the supported characteristic name
- `esp_err_t` ESP_ERR_INVALID_ARG - invalid argument of function
- `esp_err_t` ESP_ERR_NOT_FOUND - the characteristic (cid) not found
- `esp_err_t` ESP_FAIL - unknown error during lookup table processing

Parameters

- [in] `cid`: characteristic id
- `param_info`: pointer to pointer of characteristic data.

The function gets information about supported characteristic defined as cid. It will check if characteristic is supported and returns its description.

```
esp_err_t mbc_master_get_parameter(uint16_t cid, char *name, uint8_t *value, uint8_t *type)
```

Read parameter from modbus slave device whose name is defined by name and has cid. The additional data for request is taken from parameter description (lookup) table.

Return

- `esp_err_t` ESP_OK - request was successful and value buffer contains representation of actual parameter data from slave
- `esp_err_t` ESP_ERR_INVALID_ARG - invalid argument of function or parameter descriptor
- `esp_err_t` ESP_ERR_INVALID_RESPONSE - an invalid response from slave
- `esp_err_t` ESP_ERR_INVALID_STATE - invalid state during data processing or allocation failure
- `esp_err_t` ESP_ERR_TIMEOUT - operation timed out and no response from slave
- `esp_err_t` ESP_ERR_NOT_SUPPORTED - the request command is not supported by slave
- `esp_err_t` ESP_ERR_NOT_FOUND - the parameter is not found in the parameter description table
- `esp_err_t` ESP_FAIL - slave returned an exception or other failure

Parameters

- [in] `cid`: id of the characteristic for parameter
- [in] `name`: pointer into string name (key) of parameter (null terminated)
- [out] `value`: pointer to data buffer of parameter
- [out] `type`: parameter type associated with the name returned from parameter description table.

The function reads data of characteristic defined in parameters from Modbus slave device and returns its data. The additional data for request is taken from parameter description table.

```
esp_err_t mbc_master_set_parameter(uint16_t cid, char *name, uint8_t *value, uint8_t *type)
```

Set characteristic's value defined as a name and cid parameter. The additional data for cid parameter request is taken from master parameter lookup table.

Return

- `esp_err_t` ESP_OK - request was successful and value was saved in the slave device registers
- `esp_err_t` ESP_ERR_INVALID_ARG - invalid argument of function or parameter descriptor
- `esp_err_t` ESP_ERR_INVALID_RESPONSE - an invalid response from slave during processing of parameter
- `esp_err_t` ESP_ERR_INVALID_STATE - invalid state during data processing or allocation failure
- `esp_err_t` ESP_ERR_TIMEOUT - operation timed out and no response from slave
- `esp_err_t` ESP_ERR_NOT_SUPPORTED - the request command is not supported by slave
- `esp_err_t` ESP_FAIL - slave returned an exception or other failure

Parameters

- [in] `cid`: id of the characteristic for parameter

- [in] name: pointer into string name (key) of parameter (null terminated)
- [out] value: pointer to data buffer of parameter (actual representation of json value field in binary form)
- [out] type: pointer to parameter type associated with the name returned from parameter lookup table.

The function writes characteristic's value defined as a name and cid parameter in corresponded slave device. The additional data for parameter request is taken from master parameter description table.

Application Example

The examples below use the FreeModbus library port for serial slave and master implementation accordingly. The selection of stack is performed through KConfig menu "Selection of Modbus stack support mode" and related configuration keys.

[protocols/modbus/serial/mb_slave](#)

[protocols/modbus/serial/mb_master](#)

Please refer to the specific example README.md for details.

Modbus Readme File

Modbus assumptions, dependencies and constraints

1. Current implementation of Modbus has limitation: maximum frame size should be less than 120 bytes (default UART receive FIFO full threshold value). This limitation is removed in ESP-IDF v4.2.
2. The only one instance of Modbus port can be initialized at the same time.

2.4.11 ESP WebSocket Client

Overview

The ESP WebSocket client is an implementation of [WebSocket protocol client](#) for ESP32

Features

- supports WebSocket over TCP, SSL with mbedtls
- Easy to setup with URI
- Multiple instances (Multiple clients in one application)

Configuration

URI

- Supports `ws`, `wss` schemes
- WebSocket samples:
 - `ws://websocket.org`: WebSocket over TCP, default port 80
 - `wss://websocket.org`: WebSocket over SSL, default port 443
- Minimal configurations:

```
const esp_websocket_client_config_t ws_cfg = {
    .uri = "ws://websocket.org",
};
```

- If there are any options related to the URI in `esp_websocket_client_config_t`, the option defined by the URI will be overridden. Sample:

```
const esp_websocket_client_config_t ws_cfg = {
    .uri = "ws://websocket.org:123",
    .port = 4567,
};
//WebSocket client will connect to websocket.org using port 4567
```

SSL

- Get certificate from server, example: `websocket.org openssl s_client -showcerts -connect websocket.org:443 </dev/null 2>/dev/null|openssl x509 -outform PEM >websocket_org.pem`
- Configuration:

```
const esp_websocket_client_config_t ws_cfg = {
    .uri = "wss://websocket.org",
    .cert_pem = (const char *)websocket_org_pem_start,
};
```

For more options on `esp_websocket_client_config_t`, please refer to API reference below

Application Example

Simple WebSocket example that uses `esp_websocket_client` to establish a websocket connection and send/receive data with the [websocket.org](https://www.websocket.org) Server: [protocols/websocket](https://www.websocket.org/protocols/websocket).

API Reference

Header File

- [esp_websocket_client/include/esp_websocket_client.h](#)

Functions

esp_websocket_client_handle_t **esp_websocket_client_init** (**const** *esp_websocket_client_config_t* **config*)

Start a WebSocket session This function must be the first function to call, and it returns a `esp_websocket_client_handle_t` that you must use as input to other functions in the interface. This call MUST have a corresponding call to `esp_websocket_client_destroy` when the operation is complete.

Return

- `esp_websocket_client_handle_t`
- NULL if any errors

Parameters

- [in] *config*: The configuration

esp_err_t **esp_websocket_client_set_uri** (*esp_websocket_client_handle_t* *client*, **const** **char** **uri*)

Set URL for client, when performing this behavior, the options in the URL will replace the old ones Must stop the WebSocket client before set URI if the client has been connected.

Return `esp_err_t`

Parameters

- [in] *client*: The client
- [in] *uri*: The uri

esp_err_t **esp_websocket_client_start** (*esp_websocket_client_handle_t* *client*)

Open the WebSocket connection.

Return `esp_err_t`

Parameters

- [in] `client`: The client

`esp_err_t esp_websocket_client_stop(esp_websocket_client_handle_t client)`

Close the WebSocket connection.

Notes:

- Cannot be called from the websocket event handler

Return `esp_err_t`

Parameters

- [in] `client`: The client

`esp_err_t esp_websocket_client_destroy(esp_websocket_client_handle_t client)`

Destroy the WebSocket connection and free all resources. This function must be the last function to call for an session. It is the opposite of the `esp_websocket_client_init` function and must be called with the same handle as input that a `esp_websocket_client_init` call returned. This might close all connections this handle has used.

Notes:

- Cannot be called from the websocket event handler

Return `esp_err_t`

Parameters

- [in] `client`: The client

`int esp_websocket_client_send(esp_websocket_client_handle_t client, const char *data, int len, TickType_t timeout)`

Generic write data to the WebSocket connection; defaults to binary send.

Return

- Number of data was sent
- (-1) if any errors

Parameters

- [in] `client`: The client
- [in] `data`: The data
- [in] `len`: The length
- [in] `timeout`: Write data timeout in RTOS ticks

`int esp_websocket_client_send_bin(esp_websocket_client_handle_t client, const char *data, int len, TickType_t timeout)`

Write binary data to the WebSocket connection (data send with WS OPCODE=02, i.e. binary)

Return

- Number of data was sent
- (-1) if any errors

Parameters

- [in] `client`: The client
- [in] `data`: The data
- [in] `len`: The length
- [in] `timeout`: Write data timeout in RTOS ticks

`int esp_websocket_client_send_text(esp_websocket_client_handle_t client, const char *data, int len, TickType_t timeout)`

Write textual data to the WebSocket connection (data send with WS OPCODE=01, i.e. text)

Return

- Number of data was sent
- (-1) if any errors

Parameters

- [in] `client`: The client
- [in] `data`: The data
- [in] `len`: The length
- [in] `timeout`: Write data timeout in RTOS ticks

bool **esp_websocket_client_is_connected** (*esp_websocket_client_handle_t* client)

Check the WebSocket connection status.

Return

- true
- false

Parameters

- [in] client: The client handle

esp_err_t **esp_websocket_register_events** (*esp_websocket_client_handle_t* client, *esp_websocket_event_id_t* event, *esp_event_handler_t* event_handler, void *event_handler_arg)

Register the WebSocket Events.

Return esp_err_t

Parameters

- client: The client handle
- event: The event id
- event_handler: The callback function
- event_handler_arg: User context

Structures

struct esp_websocket_event_data_t

WebSocket event data.

Public Members

const char *data_ptr

Data pointer

int data_len

Data length

uint8_t op_code

Received opcode

esp_websocket_client_handle_t **client**

esp_websocket_client_handle_t context

void *user_context

user_data context, from *esp_websocket_client_config_t* user_data

int payload_len

Total payload length, payloads exceeding buffer will be posted through multiple events

int payload_offset

Actual offset for the data associated with this event

struct esp_websocket_client_config_t

WebSocket client setup configuration.

Public Members

const char *uri

WebSocket URI, the information on the URI can be overrides the other fields below, if any

const char *host

Domain or IP as string

int port

Port to connect, default depend on esp_websocket_transport_t (80 or 443)

const char *username
Using for Http authentication - Not supported for now

const char *password
Using for Http authentication - Not supported for now

const char *path
HTTP Path, if not set, default is /

bool **disable_auto_reconnect**
Disable the automatic reconnect function when disconnected

void ***user_context**
HTTP user data context

int **task_prio**
Websocket task priority

int **task_stack**
Websocket task stack

int **buffer_size**
Websocket buffer size

const char *cert_pem
SSL Certification, PEM format as string, if the client requires to verify server

esp_websocket_transport_t **transport**
Websocket transport type, see `esp_websocket_transport_t`

char ***subprotocol**
Websocket subprotocol

char ***user_agent**
Websocket user-agent

char ***headers**
Websocket additional headers

bool **keep_alive_enable**
Enable keep-alive timeout

int **keep_alive_idle**
Keep-alive idle time. Default is 5 (second)

int **keep_alive_interval**
Keep-alive interval time. Default is 5 (second)

int **keep_alive_count**
Keep-alive packet retry send count. Default is 3 counts

Type Definitions

```
typedef struct esp_websocket_client *esp_websocket_client_handle_t
```

Enumerations

```
enum esp_websocket_event_id_t
```

Websocket Client events id.

Values:

WEBSOCKET_EVENT_ANY = -1

WEBSOCKET_EVENT_ERROR = 0

This event occurs when there are any errors during execution

WEBSOCKET_EVENT_CONNECTED

Once the Websocket has been connected to the server, no data exchange has been performed

WEBSOCKET_EVENT_DISCONNECTED

The connection has been disconnected

WEBSOCKET_EVENT_DATA

When receiving data from the server, possibly multiple portions of the packet

WEBSOCKET_EVENT_MAX**enum esp_websocket_transport_t**

Websocket Client transport.

Values:

WEBSOCKET_TRANSPORT_UNKNOWN = 0x0

Transport unknown

WEBSOCKET_TRANSPORT_OVER_TCP

Transport over tcp

WEBSOCKET_TRANSPORT_OVER_SSL

Transport over ssl

2.4.12 ESP Serial Slave Link

Overview

Espressif provides several chips that can work as slaves. These slave devices rely on some common buses, and have their own communication protocols over those buses. The *esp_serial_slave_link* component is designed for the master to communicate with ESP slave devices through those protocols over the bus drivers.

After an *esp_serial_slave_link* device is initialized properly, the application can use it to communicate with the ESP slave devices conveniently.

For more details about ESP32 SDIO slave protocol, see document [Communication with ESP SDIO Slave](#).

Terminology

- ESSL: Abbreviation for ESP Serial Slave Link, the component described by this document.
- Master: The device running the *esp_serial_slave_link* component.
- ESSL device: a virtual device on the master associated with an ESP slave device. The device context has the knowledge of the slave protocol above the bus, relying on some bus drivers to communicate with the slave.
- ESSL device handle: a handle to ESSL device context containing the configuration, status and data required by the ESSL component. The context stores the driver configurations, communication state, data shared by master and slave, etc.
The context should be initialized before it is used, and get deinitialized if not used any more. The master application operates on the ESSL device through this handle.
- ESP slave: the slave device connected to the bus, which ESSL component is designed to communicate with.
- Bus: The bus over which the master and the slave communicate with each other.
- Slave protocol: The special communication protocol specified by Espressif HW/SW over the bus.
- TX buffer num: a counter, which is on the slave and can be read by the master, indicates the accumulated buffer numbers that the slave has loaded to the hardware to receive data from the master.
- RX data size: a counter, which is on the slave and can be read by the master, indicates the accumulated data size that the slave has loaded to the hardware to send to the master.

Services provided by ESP slave

There are some common services provided by the Espressif slaves:

1. Tohost Interrupts: The slave can inform the master about certain events by the interrupt line.
2. Frhost Interrupts: The master can inform the slave about certain events.

3. Tx FIFO (master to slave): the slave can send data in stream to the master. The SDIO slave can also indicate it has new data to send to master by the interrupt line.
The slave updates the TX buffer num to inform the master how much data it can receive, and the master then read the TX buffer num, and take off the used buffer number to know how many buffers are remaining.
4. Rx FIFO (slave to master): the slave can receive data from the master in units of receiving buffers.
The slave updates the RX data size to inform the master how much data it has prepared to send, and then the master read the data size, and take off the data length it has already received to know how many data is remaining.
5. Shared registers: the master can read some part of the registers on the slave, and also write these registers to let the slave read.

Initialization of ESP SDIO Slave Link

The ESP SDIO slave link (ESSL SDIO) devices relies on the sdmmc component. The ESSL device should be initialized as below:

1. Initialize a sdmmc card (see :doc:` Document of SDMMC driver </api-reference/storage/sdmmc>`) structure.
2. Call `sdmmc_card_init()` to initialize the card.
3. Initialize the ESSL device with `essl_sdio_config_t`. The `card` member should be the `sdmmc_card_t` got in step 2, and the `recv_buffer_size` member should be filled correctly according to pre-negotiated value.
4. Call `essl_init()` to do initialization of the SDIO part.
5. Call `essl_wait_for_ready()` to wait for the slave to be ready.

APIs

After the initialization process above is performed, you can call the APIs below to make use of the services provided by the slave:

Interrupts

1. Call `essl_get_intr_ena()` to know which events will trigger the interrupts to the master.
2. Call `essl_set_intr_ena()` to set the events that will trigger interrupts to the master.
3. Call `essl_wait_int()` to wait until interrupt from the slave, or timeout.
4. When interrupt is triggered, call `essl_get_intr()` to know which events are active, and call `essl_clear_intr()` to clear them.
5. Call `essl_send_slave_intr()` to trigger general purpose interrupt of the slave.

TX FIFO

1. Call `essl_get_tx_buffer_num()` to know how many buffers the slave has prepared to receive data from the master. This is optional. The master will poll `tx_buffer_num` when it try to send packets to the slave, until the slave has enough buffer or timeout.
2. Call `essl_send_packet()` to send data to the slave.

RX FIFO

1. Call `essl_get_rx_data_size()` to know how many data the slave has prepared to send to the master. This is optional. When the master tries to receive data from the slave, it will update the `rx_data_size` for once, if the current `rx_data_size` is shorter than the buffer size the master prepared to receive. And it may poll the `rx_data_size` if the `rx_data_size` keeps 0, until timeout.
2. Call `essl_get_packet()` to receive data from the slave.

Reset counters (Optional) Call `essl_reset_cnt()` to reset the internal counter if you find the slave has reset its counter.

Application Example

The example below shows how ESP32 SDIO host and slave communicate with each other. The host use the ESSL SDIO.

[peripherals/sdio](#).

Please refer to the specific example README.md for details.

API Reference

Header File

- [esp_serial_slave_link/include/esp_serial_slave_link/essl.h](#)

Functions

[esp_err_t](#) **essl_init** ([essl_handle_t](#) handle, uint32_t wait_ms)

Initialize the slave.

Return ESP_OK if success, or other value returned from lower layer `init`.

Parameters

- handle: Handle of a `essl` device.
- wait_ms: Millisecond to wait before timeout, will not wait at all if set to 0-9.

[esp_err_t](#) **essl_wait_for_ready** ([essl_handle_t](#) handle, uint32_t wait_ms)

Wait for interrupt of a ESP32 slave device.

Return

- ESP_OK if success
- One of the error codes from SDMMC host controller

Parameters

- handle: Handle of a `essl` device.
- wait_ms: Millisecond to wait before timeout, will not wait at all if set to 0-9.

[esp_err_t](#) **essl_get_tx_buffer_num** ([essl_handle_t](#) handle, uint32_t *out_tx_num, uint32_t wait_ms)

Get buffer num for the host to send data to the slave. The buffers are size of `buffer_size`.

Return

- ESP_OK Success
- One of the error codes from SDMMC host controller

Parameters

- handle: Handle of a `essl` device.
- out_tx_num: Output of buffer num that host can send data to ESP32 slave.
- wait_ms: Millisecond to wait before timeout, will not wait at all if set to 0-9.

[esp_err_t](#) **essl_get_rx_data_size** ([essl_handle_t](#) handle, uint32_t *out_rx_size, uint32_t wait_ms)

Get amount of data the ESP32 slave preparing to send to host.

Return

- ESP_OK Success
- One of the error codes from SDMMC host controller

Parameters

- handle: Handle of a `essl` device.
- out_rx_size: Output of data size to read from slave.
- wait_ms: Millisecond to wait before timeout, will not wait at all if set to 0-9.

[esp_err_t](#) **essl_reset_cnt** ([essl_handle_t](#) handle)

Reset the counters of this component. Usually you don't need to do this unless you know the slave is reset.

Parameters

- handle: Handle of a `essl` device.

esp_err_t **essl_send_packet** (*essl_handle_t* handle, const void *start, size_t length, uint32_t wait_ms)

Send a packet to the ESP32 slave. The slave receive the packet into buffers whose size is `buffer_size` (configured during initialization).

Return

- ESP_OK Success
- ESP_ERR_TIMEOUT No buffer to use, or error from SDMMC host controller
- One of the error codes from SDMMC host controller

Parameters

- handle: Handle of a `essl` device.
- start: Start address of the packet to send
- length: Length of data to send, if the packet is over-size, the it will be divided into blocks and hold into different buffers automatically.
- wait_ms: Millisecond to wait before timeout, will not wait at all if set to 0-9.

esp_err_t **essl_get_packet** (*essl_handle_t* handle, void *out_data, size_t size, size_t *out_length, uint32_t wait_ms)

Get a packet from ESP32 slave.

Return

- ESP_OK Success, all the data are read from the slave.
- ESP_ERR_NOT_FINISHED Read success, while there' re data remaining.
- One of the error codes from SDMMC host controller

Parameters

- handle: Handle of a `essl` device.
- [out] out_data: Data output address
- size: The size of the output buffer, if the buffer is smaller than the size of data to receive from slave, the driver returns ESP_ERR_NOT_FINISHED
- [out] out_length: Output of length the data actually received from slave.
- wait_ms: Millisecond to wait before timeout, will not wait at all if set to 0-9.

esp_err_t **essl_write_reg** (*essl_handle_t* handle, uint8_t addr, uint8_t value, uint8_t *value_o, uint32_t wait_ms)

Write general purpose R/W registers (8-bit) of ESP32 slave.

Note sdio 28-31 are reserved, the lower API helps to skip.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Address not valid.
- One of the error codes from SDMMC host controller

Parameters

- handle: Handle of a `essl` device.
- addr: Address of register to write. Valid address: 0-59.
- value: Value to write to the register.
- value_o: Output of the returned written value.
- wait_ms: Millisecond to wait before timeout, will not wait at all if set to 0-9.

esp_err_t **essl_read_reg** (*essl_handle_t* handle, uint8_t addr, uint8_t *value_o, uint32_t wait_ms)

Read general purpose R/W registers (8-bit) of ESP32 slave.

Return

- ESP_OK Success
- ESP_ERR_INVALID_ARG Address not valid.
- One of the error codes from SDMMC host controller

Parameters

- handle: Handle of a `essl` device.
- add: Address of register to read. Valid address: 0-27, 32-63 (28-31 reserved, return interrupt bits on read).
- value_o: Output value read from the register.
- wait_ms: Millisecond to wait before timeout, will not wait at all if set to 0-9.

esp_err_t **essl_wait_int** (*essl_handle_t* handle, uint32_t wait_ms)

wait for an interrupt of the slave

Return

- ESP_ERR_NOT_SUPPORTED Currently our driver doesnot support SDIO with SPI interface.
- ESP_OK If interrupt triggered.
- ESP_ERR_TIMEOUT No interrupts before timeout.

Parameters

- `handle`: Handle of a `essl` device.
- `wait_ms`: Millisecond to wait before timeout, will not wait at all if set to 0-9.

esp_err_t **essl_clear_intr** (*essl_handle_t* `handle`, *uint32_t* `intr_mask`, *uint32_t* `wait_ms`)

Clear interrupt bits of ESP32 slave. All the bits set in the mask will be cleared, while other bits will stay the same.

Return

- ESP_OK Success
- One of the error codes from SDMMC host controller

Parameters

- `handle`: Handle of a `essl` device.
- `intr_mask`: Mask of interrupt bits to clear.
- `wait_ms`: Millisecond to wait before timeout, will not wait at all if set to 0-9.

esp_err_t **essl_get_intr** (*essl_handle_t* `handle`, *uint32_t* *`intr_raw`, *uint32_t* *`intr_st`, *uint32_t* `wait_ms`)

Get interrupt bits of ESP32 slave.

Return

- ESP_OK Success
- ESP_INVALID_ARG if both `intr_raw` and `intr_st` are NULL.
- One of the error codes from SDMMC host controller

Parameters

- `handle`: Handle of a `essl` device.
- `intr_raw`: Output of the raw interrupt bits. Set to NULL if only masked bits are read.
- `intr_st`: Output of the masked interrupt bits. set to NULL if only raw bits are read.
- `wait_ms`: Millisecond to wait before timeout, will not wait at all if set to 0-9.

esp_err_t **essl_set_intr_ena** (*essl_handle_t* `handle`, *uint32_t* `ena_mask`, *uint32_t* `wait_ms`)

Set interrupt enable bits of ESP32 slave. The slave only sends interrupt on the line when there is a bit both the raw status and the enable are set.

Return

- ESP_OK Success
- One of the error codes from SDMMC host controller

Parameters

- `handle`: Handle of a `essl` device.
- `ena_mask`: Mask of the interrupt bits to enable.
- `wait_ms`: Millisecond to wait before timeout, will not wait at all if set to 0-9.

esp_err_t **essl_get_intr_ena** (*essl_handle_t* `handle`, *uint32_t* *`ena_mask_o`, *uint32_t* `wait_ms`)

Get interrupt enable bits of ESP32 slave.

Return

- ESP_OK Success
- One of the error codes from SDMMC host controller

Parameters

- `handle`: Handle of a `essl` device.
- `ena_mask_o`: Output of interrupt bit enable mask.
- `wait_ms`: Millisecond to wait before timeout, will not wait at all if set to 0-9.

esp_err_t **essl_send_slave_intr** (*essl_handle_t* `handle`, *uint32_t* `intr_mask`, *uint32_t* `wait_ms`)

Send interrupts to slave. Each bit of the interrupt will be triggered.

Return

- ESP_OK Success
- One of the error codes from SDMMC host controller

Parameters

- `handle`: Handle of a `essl` device.
- `intr_mask`: Mask of interrupt bits to send to slave.
- `wait_ms`: Millisecond to wait before timeout, will not wait at all if set to 0-9.

Macros

ESP_ERR_NOT_FINISHED

There is still remaining data.

Type Definitions

```
typedef struct essl_dev_t *essl_handle_t
```

Handle of an ESSL device.

Header File

- `esp_serial_slave_link/include/esp_serial_slave_link/essl_sdio.h`

Functions

```
esp_err_t essl_sdio_init_dev (essl_handle_t *out_handle, const essl_sdio_config_t *config)
```

Initialize the ESSL SDIO device and get its handle.

Return

- `ESP_OK`: on success
- `ESP_ERR_NO_MEM`: memory exhausted.

Parameters

- `out_handle`: Output of the handle.
- `config`: Configuration for the ESSL SDIO device.

```
esp_err_t essl_sdio_deinit_dev (essl_handle_t handle)
```

Deinitialize and free the space used by the ESSL SDIO device.

Return

- `ESP_OK`: on success
- `ESP_ERR_INVALID_ARG`: wrong handle passed

Parameters

- `handle`: Handle of the ESSL SDIO device to deinit.

Structures

```
struct essl_sdio_config_t
```

Configuration for the `essl` SDIO device.

Public Members

```
sdmmc_card_t *card
```

The initialized `sdmmc` card pointer of the slave.

```
int recv_buffer_size
```

The pre-negotiated `recv` buffer size used by both the host and the slave.

Code examples for this API section are provided in the [protocols](#) directory of ESP-IDF examples.

2.4.13 IP Network Layer

Documentation for IP Network Layer protocols (below the Application Protocol layer) are provided in [Networking APIs](#).

2.5 Provisioning API

2.5.1 Protocol Communication

Overview

Protocol Communication (protocomm) component manages secure sessions and provides framework for multiple transports. The application can also use protocomm layer directly to have application specific extensions for the provisioning (or non-provisioning) use cases.

Following features are available for provisioning :

- **Communication security at application level -**
 - protocomm_security0 (no security)
 - protocomm_security1 (curve25519 key exchange + AES-CTR encryption)
- Proof-of-possession (support with protocomm_security1 only)

Protocomm internally uses protobuf (protocol buffers) for secure session establishment. Though users can implement their own security (even without using protobuf). One can even use protocomm without any security layer.

Protocomm provides framework for various transports - WiFi (SoftAP+HTTPD), BLE, console - in which case the handler invocation is automatically taken care of on the device side (see Transport Examples below for code snippets).

Note that the client still needs to establish session (only for protocomm_security1) by performing the two way handshake. See [Unified Provisioning](#) for more details about the secure handshake logic.

Transport Example (SoftAP + HTTP) with Security 1

For complete example see [provisioning/softap_prov](#)

```

/* Endpoint handler to be registered with protocomm.
 * This simply echoes back the received data. */
esp_err_t echo_req_handler (uint32_t session_id,
                            const uint8_t *inbuf, ssize_t inlen,
                            uint8_t **outbuf, ssize_t *outlen,
                            void *priv_data)
{
    /* Session ID may be used for persistence */
    printf("Session ID : %d", session_id);

    /* Echo back the received data */
    *outlen = inlen;          /* Output data length updated */
    *outbuf = malloc(inlen); /* This will be deallocated outside */
    memcpy(*outbuf, inbuf, inlen);

    /* Private data that was passed at the time of endpoint creation */
    uint32_t *priv = (uint32_t *) priv_data;
    if (priv) {
        printf("Private data : %d", *priv);
    }

    return ESP_OK;
}

/* Example function for launching a protocomm instance over HTTP */
protocomm_t *start_pc(const char *pop_string)
{
    protocomm_t *pc = protocomm_new();

    /* Config for protocomm_httpd_start() */
    protocomm_httpd_config_t pc_config = {

```

(continues on next page)

(continued from previous page)

```

        .data = {
        .config = PROTOCOMM_HTTPD_DEFAULT_CONFIG()
        }
};

/* Start protocomm server on top of HTTP */
protocomm_httpd_start(pc, &pc_config);

/* Create Proof of Possession object from pop_string. It must be valid
 * throughout the scope of protocomm endpoint. This need not be
↪static,
 * ie. could be dynamically allocated and freed at the time of
↪endpoint
↪removal */
const static protocomm_security_pop_t pop_obj = {
    .data = (const uint8_t *) strdup(pop_string),
    .len = strlen(pop_string)
};

/* Set security for communication at application level. Just like for
 * request handlers, setting security creates an endpoint and
↪registers
 * the handler provided by protocomm_security1. One can similarly use
 * protocomm_security0. Only one type of security can be set for a
 * protocomm instance at a time. */
protocomm_set_security(pc, "security_endpoint", &protocomm_security1,
↪&pop_obj);

/* Private data passed to the endpoint must be valid throughout the
↪scope
 * of protocomm endpoint. This need not be static, ie. could be
↪dynamically
↪allocated and freed at the time of endpoint removal */
static uint32_t priv_data = 1234;

/* Add a new endpoint for the protocomm instance, identified by a
↪unique name
 * and register a handler function along with private data to be
↪passed at the
 * time of handler execution. Multiple endpoints can be added as long
↪as they
 * are identified by unique names */
protocomm_add_endpoint(pc, "echo_req_endpoint",
    echo_req_handler, (void *) &priv_data);

return pc;
}

/* Example function for stopping a protocomm instance */
void stop_pc(protocomm_t *pc)
{
    /* Remove endpoint identified by it's unique name */
    protocomm_remove_endpoint(pc, "echo_req_endpoint");

    /* Remove security endpoint identified by it's name */
    protocomm_unset_security(pc, "security_endpoint");

    /* Stop HTTP server */
    protocomm_httpd_stop(pc);

    /* Delete (deallocate) the protocomm instance */
    protocomm_delete(pc);
}

```

(continues on next page)

(continued from previous page)

}

Transport Example (BLE) with Security 0

For complete example see [provisioning/ble_prov](#)

```

/* Example function for launching a secure protocomm instance over BLE */
protocomm_t *start_pc()
{
    protocomm_t *pc = protocomm_new();

    /* Endpoint UUIDs */
    protocomm_ble_name_uuid_t nu_lookup_table[] = {
        {"security_endpoint", 0xFF51},
        {"echo_req_endpoint", 0xFF52}
    };

    /* Config for protocomm_ble_start() */
    protocomm_ble_config_t config = {
        .service_uuid = {
            /* LSB <-----> MSB */
            0xfb, 0x34, 0x9b, 0x5f, 0x80, 0x00, 0x00, 0x80,
            0x00, 0x10, 0x00, 0x00, 0xFF, 0xFF, 0x00, 0x00,
        },
        .nu_lookup_count = sizeof(nu_lookup_table)/sizeof(nu_lookup_
→table[0]),
        .nu_lookup = nu_lookup_table
    };

    /* Start protocomm layer on top of BLE */
    protocomm_ble_start(pc, &config);

    /* For protocomm_security0, Proof of Possession is not used, and can
→be kept NULL */
    protocomm_set_security(pc, "security_endpoint", &protocomm_security0,
→NULL);
    protocomm_add_endpoint(pc, "echo_req_endpoint", echo_req_handler,
→NULL);
    return pc;
}

/* Example function for stopping a protocomm instance */
void stop_pc(protocomm_t *pc)
{
    protocomm_remove_endpoint(pc, "echo_req_endpoint");
    protocomm_unset_security(pc, "security_endpoint");

    /* Stop BLE protocomm service */
    protocomm_ble_stop(pc);

    protocomm_delete(pc);
}

```

API Reference

Header File

- [protocomm/include/common/protocomm.h](#)

Functions

protocomm_t ***protocomm_new** (void)

Create a new protocomm instance.

This API will return a new dynamically allocated protocomm instance with all elements of the *protocomm_t* structure initialized to NULL.

Return

- *protocomm_t** : On success
- NULL : No memory for allocating new instance

void **protocomm_delete** (*protocomm_t* *pc)

Delete a protocomm instance.

This API will deallocate a protocomm instance that was created using `protocomm_new()`.

Parameters

- [in] pc: Pointer to the protocomm instance to be deleted

esp_err_t **protocomm_add_endpoint** (*protocomm_t* *pc, const char *ep_name, *protocomm_req_handler_t* h, void *priv_data)

Add endpoint request handler for a protocomm instance.

This API will bind an endpoint handler function to the specified endpoint name, along with any private data that needs to be pass to the handler at the time of call.

Note

- An endpoint must be bound to a valid protocomm instance, created using `protocomm_new()`.
- This function internally calls the registered `add_endpoint()` function of the selected transport which is a member of the *protocomm_t* instance structure.

Return

- ESP_OK : Success
- ESP_FAIL : Error adding endpoint / Endpoint with this name already exists
- ESP_ERR_NO_MEM : Error allocating endpoint resource
- ESP_ERR_INVALID_ARG : Null instance/name/handler arguments

Parameters

- [in] pc: Pointer to the protocomm instance
- [in] ep_name: Endpoint identifier(name) string
- [in] h: Endpoint handler function
- [in] priv_data: Pointer to private data to be passed as a parameter to the handler function on call. Pass NULL if not needed.

esp_err_t **protocomm_remove_endpoint** (*protocomm_t* *pc, const char *ep_name)

Remove endpoint request handler for a protocomm instance.

This API will remove a registered endpoint handler identified by an endpoint name.

Note

- This function internally calls the registered `remove_endpoint()` function which is a member of the *protocomm_t* instance structure.

Return

- ESP_OK : Success
- ESP_ERR_NOT_FOUND : Endpoint with specified name doesn't exist
- ESP_ERR_INVALID_ARG : Null instance/name arguments

Parameters

- [in] pc: Pointer to the protocomm instance
- [in] ep_name: Endpoint identifier(name) string

esp_err_t **protocomm_open_session** (*protocomm_t* *pc, uint32_t session_id)

Allocates internal resources for new transport session.

Note

- An endpoint must be bound to a valid protocomm instance, created using `protocomm_new()`.

Return

- ESP_OK : Request handled successfully

- `ESP_ERR_NO_MEM` : Error allocating internal resource
- `ESP_ERR_INVALID_ARG` : Null instance/name arguments

Parameters

- `[in] pc`: Pointer to the `protocomm` instance
- `[in] session_id`: Unique ID for a communication session

`esp_err_t protocomm_close_session(protocomm_t *pc, uint32_t session_id)`

Frees internal resources used by a transport session.

Note

- An endpoint must be bound to a valid `protocomm` instance, created using `protocomm_new()`.

Return

- `ESP_OK` : Request handled successfully
- `ESP_ERR_INVALID_ARG` : Null instance/name arguments

Parameters

- `[in] pc`: Pointer to the `protocomm` instance
- `[in] session_id`: Unique ID for a communication session

`esp_err_t protocomm_req_handle(protocomm_t *pc, const char *ep_name, uint32_t session_id, const uint8_t *inbuf, ssize_t inlen, uint8_t **outbuf, ssize_t *outlen)`

Calls the registered handler of an endpoint session for processing incoming data and generating the response.

Note

- An endpoint must be bound to a valid `protocomm` instance, created using `protocomm_new()`.
- Resulting output buffer must be deallocated by the caller.

Return

- `ESP_OK` : Request handled successfully
- `ESP_FAIL` : Internal error in execution of registered handler
- `ESP_ERR_NO_MEM` : Error allocating internal resource
- `ESP_ERR_NOT_FOUND` : Endpoint with specified name doesn't exist
- `ESP_ERR_INVALID_ARG` : Null instance/name arguments

Parameters

- `[in] pc`: Pointer to the `protocomm` instance
- `[in] ep_name`: Endpoint identifier(name) string
- `[in] session_id`: Unique ID for a communication session
- `[in] inbuf`: Input buffer contains input request data which is to be processed by the registered handler
- `[in] inlen`: Length of the input buffer
- `[out] outbuf`: Pointer to internally allocated output buffer, where the resulting response data output from the registered handler is to be stored
- `[out] outlen`: Buffer length of the allocated output buffer

`esp_err_t protocomm_set_security(protocomm_t *pc, const char *ep_name, const protocomm_security_t *sec, const protocomm_security_pop_t *pop)`

Add endpoint security for a `protocomm` instance.

This API will bind a security session establisher to the specified endpoint name, along with any proof of possession that may be required for authenticating a session client.

Note

- An endpoint must be bound to a valid `protocomm` instance, created using `protocomm_new()`.
- The choice of security can be any `protocomm_security_t` instance. Choices `protocomm_security0` and `protocomm_security1` are readily available.

Return

- `ESP_OK` : Success
- `ESP_FAIL` : Error adding endpoint / Endpoint with this name already exists
- `ESP_ERR_INVALID_STATE` : Security endpoint already set
- `ESP_ERR_NO_MEM` : Error allocating endpoint resource
- `ESP_ERR_INVALID_ARG` : Null instance/name/handler arguments

Parameters

- [in] `pc`: Pointer to the protocomm instance
- [in] `ep_name`: Endpoint identifier(name) string
- [in] `sec`: Pointer to endpoint security instance
- [in] `pop`: Pointer to proof of possession for authenticating a client

`esp_err_t protocomm_unset_security(protocomm_t *pc, const char *ep_name)`
Remove endpoint security for a protocomm instance.

This API will remove a registered security endpoint identified by an endpoint name.

Return

- `ESP_OK` : Success
- `ESP_ERR_NOT_FOUND` : Endpoint with specified name doesn't exist
- `ESP_ERR_INVALID_ARG` : Null instance/name arguments

Parameters

- [in] `pc`: Pointer to the protocomm instance
- [in] `ep_name`: Endpoint identifier(name) string

`esp_err_t protocomm_set_version(protocomm_t *pc, const char *ep_name, const char *version)`
Set endpoint for version verification.

This API can be used for setting an application specific protocol version which can be verified by clients through the endpoint.

Note

- An endpoint must be bound to a valid protocomm instance, created using `protocomm_new()`.

Return

- `ESP_OK` : Success
- `ESP_FAIL` : Error adding endpoint / Endpoint with this name already exists
- `ESP_ERR_INVALID_STATE` : Version endpoint already set
- `ESP_ERR_NO_MEM` : Error allocating endpoint resource
- `ESP_ERR_INVALID_ARG` : Null instance/name/handler arguments

Parameters

- [in] `pc`: Pointer to the protocomm instance
- [in] `ep_name`: Endpoint identifier(name) string
- [in] `version`: Version identifier(name) string

`esp_err_t protocomm_unset_version(protocomm_t *pc, const char *ep_name)`
Remove version verification endpoint from a protocomm instance.

This API will remove a registered version endpoint identified by an endpoint name.

Return

- `ESP_OK` : Success
- `ESP_ERR_NOT_FOUND` : Endpoint with specified name doesn't exist
- `ESP_ERR_INVALID_ARG` : Null instance/name arguments

Parameters

- [in] `pc`: Pointer to the protocomm instance
- [in] `ep_name`: Endpoint identifier(name) string

Type Definitions

```
typedef esp_err_t (*protocomm_req_handler_t) (uint32_t session_id, const uint8_t *inbuf,
                                             ssize_t inlen, uint8_t **outbuf, ssize_t *outlen,
                                             void *priv_data)
```

Function prototype for protocomm endpoint handler.

```
typedef struct protocomm protocomm_t
```

This structure corresponds to a unique instance of protocomm returned when the API `protocomm_new()` is called. The remaining Protocomm APIs require this object as the first parameter.

Note Structure of the protocomm object is kept private

Header File

- `protocomm/include/security/protocomm_security.h`

Structures

struct protocomm_security_pop
Proof Of Possession for authenticating a secure session.

Public Members

const uint8_t *data
Pointer to buffer containing the proof of possession data

uint16_t len
Length (in bytes) of the proof of possession data

struct protocomm_security
Protocomm security object structure.

The member functions are used for implementing secure protocomm sessions.

Note This structure should not have any dynamic members to allow re-entrancy

Public Members

int ver
Unique version number of security implementation

*esp_err_t (*init) (protocomm_security_handle_t *handle)*
Function for initializing/allocating security infrastructure

*esp_err_t (*cleanup) (protocomm_security_handle_t handle)*
Function for deallocating security infrastructure

*esp_err_t (*new_transport_session) (protocomm_security_handle_t handle, uint32_t session_id)*
Starts new secure transport session with specified ID

*esp_err_t (*close_transport_session) (protocomm_security_handle_t handle, uint32_t session_id)*
Closes a secure transport session with specified ID

*esp_err_t (*security_req_handler) (protocomm_security_handle_t handle, const protocomm_security_pop_t *pop, uint32_t session_id, const uint8_t *inbuf, ssize_t inlen, uint8_t **outbuf, ssize_t *outlen, void *priv_data)*
Handler function for authenticating connection request and establishing secure session

*esp_err_t (*encrypt) (protocomm_security_handle_t handle, uint32_t session_id, const uint8_t *inbuf, ssize_t inlen, uint8_t *outbuf, ssize_t *outlen)*
Function which implements the encryption algorithm

*esp_err_t (*decrypt) (protocomm_security_handle_t handle, uint32_t session_id, const uint8_t *inbuf, ssize_t inlen, uint8_t *outbuf, ssize_t *outlen)*
Function which implements the decryption algorithm

Type Definitions

typedef struct protocomm_security_pop protocomm_security_pop_t
Proof Of Possession for authenticating a secure session.

typedef void *protocomm_security_handle_t

typedef struct *protocomm_security* protocomm_security_t

Protocomm security object structure.

The member functions are used for implementing secure protocomm sessions.

Note This structure should not have any dynamic members to allow re-entrancy

Header File

- [protocomm/include/security/protocomm_security0.h](#)

Header File

- [protocomm/include/security/protocomm_security1.h](#)

Header File

- [protocomm/include/transports/protocomm_httpd.h](#)

Functions

esp_err_t **protocomm_httpd_start** (*protocomm_t* *pc, const *protocomm_httpd_config_t* *config)

Start HTTPD protocomm transport.

This API internally creates a framework to allow endpoint registration and security configuration for the protocomm.

Note This is a singleton. ie. Protocomm can have multiple instances, but only one instance can be bound to an HTTP transport layer.

Return

- ESP_OK : Success
- ESP_ERR_INVALID_ARG : Null arguments
- ESP_ERR_NOT_SUPPORTED : Transport layer bound to another protocomm instance
- ESP_ERR_INVALID_STATE : Transport layer already bound to this protocomm instance
- ESP_ERR_NO_MEM : Memory allocation for server resource failed
- ESP_ERR_HTTPD_* : HTTP server error on start

Parameters

- [in] pc: Protocomm instance pointer obtained from `protocomm_new()`
- [in] config: Pointer to config structure for initializing HTTP server

esp_err_t **protocomm_httpd_stop** (*protocomm_t* *pc)

Stop HTTPD protocomm transport.

This API cleans up the HTTPD transport protocomm and frees all the handlers registered with the protocomm.

Return

- ESP_OK : Success
- ESP_ERR_INVALID_ARG : Null / incorrect protocomm instance pointer

Parameters

- [in] pc: Same protocomm instance that was passed to `protocomm_httpd_start()`

Unions

union protocomm_httpd_config_data_t

#include <protocomm_httpd.h> Protocomm HTTPD Configuration Data

Public Members

void ***handle**

HTTP Server Handle, if `ext_handle_provided` is set to true

***protocomm_http_server_config_t* config**

HTTP Server Configuration, if a server is not already active

Structures**struct protocomm_http_server_config_t**

Config parameters for protocomm HTTP server.

Public Members**uint16_t port**

Port on which the HTTP server will listen

size_t stack_size

Stack size of server task, adjusted depending upon stack usage of endpoint handler

unsigned task_priority

Priority of server task

struct protocomm_httpd_config_t

Config parameters for protocomm HTTP server.

Public Members**bool ext_handle_provided**

Flag to indicate of an external HTTP Server Handle has been provided. In such as case, protocomm will use the same HTTP Server and not start a new one internally.

***protocomm_httpd_config_data_t* data**

Protocomm HTTPD Configuration Data

Macros**PROTOCOLM_HTTPD_DEFAULT_CONFIG ()****Header File**

- [protocomm/include/transport/protocomm_ble.h](#)

Functions***esp_err_t* protocomm_ble_start (*protocomm_t* *pc, const *protocomm_ble_config_t* *config)**

Start Bluetooth Low Energy based transport layer for provisioning.

Initialize and start required BLE service for provisioning. This includes the initialization for characteristics/service for BLE.

Return

- ESP_OK : Success
- ESP_FAIL : Simple BLE start error
- ESP_ERR_NO_MEM : Error allocating memory for internal resources
- ESP_ERR_INVALID_STATE : Error in ble config
- ESP_ERR_INVALID_ARG : Null arguments

Parameters

- [in] pc: Protocomm instance pointer obtained from protocomm_new()
- [in] config: Pointer to config structure for initializing BLE

***esp_err_t* protocomm_ble_stop (*protocomm_t* *pc)**

Stop Bluetooth Low Energy based transport layer for provisioning.

Stop service/task responsible for BLE based interactions for provisioning

Note You might want to optionally reclaim memory from Bluetooth. Refer to the documentation of `esp_bt_mem_release` in that case.

Return

- `ESP_OK` : Success
- `ESP_FAIL` : Simple BLE stop error
- `ESP_ERR_INVALID_ARG` : Null / incorrect protocomm instance

Parameters

- `[in] pc`: Same protocomm instance that was passed to `protocomm_ble_start()`

Structures**struct name_uuid**

This structure maps handler required by protocomm layer to UUIDs which are used to uniquely identify BLE characteristics from a smartphone or a similar client device.

Public Members

const char *name

Name of the handler, which is passed to protocomm layer

uint16_t uuid

UUID to be assigned to the BLE characteristic which is mapped to the handler

struct protocomm_ble_config

Config parameters for protocomm BLE service.

Public Members

char device_name[MAX_BLE_DEVNAME_LEN]

BLE device name being broadcast at the time of provisioning

uint8_t service_uuid[BLE_UUID128_VAL_LENGTH]

128 bit UUID of the provisioning service

ssize_t nu_lookup_count

Number of entries in the Name-UUID lookup table

***protocomm_ble_name_uuid_t* *nu_lookup**

Pointer to the Name-UUID lookup table

Macros

MAX_BLE_DEVNAME_LEN

BLE device name cannot be larger than this value 31 bytes (max scan response size) - 1 byte (length) - 1 byte (type) = 29 bytes

BLE_UUID128_VAL_LENGTH

Type Definitions

typedef struct *name_uuid* protocomm_ble_name_uuid_t

This structure maps handler required by protocomm layer to UUIDs which are used to uniquely identify BLE characteristics from a smartphone or a similar client device.

typedef struct *protocomm_ble_config* protocomm_ble_config_t

Config parameters for protocomm BLE service.

2.5.2 Unified Provisioning

Overview

Unified provisioning support in the ESP-IDF provides an extensible mechanism to the developers to configure the device with the Wi-Fi credentials and/or other custom configuration using various transports and different security schemes. Depending on the use-case it provides a complete and ready solution for Wi-Fi network provisioning along with example iOS and Android applications. Or developers can extend the device-side and phone-app side implementations to accommodate their requirements for sending additional configuration data. Following are the important features of this implementation.

1. *Extensible Protocol*: The protocol is completely flexible and it offers the ability for the developers to send custom configuration in the provisioning process. The data representation too is left to the application to decide.
2. *Transport Flexibility*: The protocol can work on Wi-Fi (SoftAP + HTTP server) or on BLE as a transport protocol. The framework provides an ability to add support for any other transport easily as long as command-response behaviour can be supported on the transport.
3. *Security Scheme Flexibility*: It's understood that each use-case may require different security scheme to secure the data that is exchanged in the provisioning process. Some applications may work with SoftAP that's WPA2 protected or BLE with "just-works" security. Or the applications may consider the transport to be insecure and may want application level security. The unified provisioning framework allows application to choose the security as deemed suitable.
4. *Compact Data Representation*: The protocol uses [Google Protobufs](#) as a data representation for session setup and Wi-Fi provisioning. They provide a compact data representation and ability to parse the data in multiple programming languages in native format. Please note that this data representation is not forced on application specific data and the developers may choose the representation of their choice.

Typical Provisioning Process

Deciding on Transport

Unified provisioning subsystem supports Wi-Fi (SoftAP+HTTP server) and BLE (GATT based) transport schemes. Following points need to be considered while selecting the best possible transport for provisioning.

1. BLE based transport has an advantage that in the provisioning process, the BLE communication channel stays intact between the device and the client. That provides reliable provisioning feedback.
2. BLE based provisioning implementation makes the user-experience better from the phone apps as on Android and iOS both, the phone app can discover and connect to the device without requiring user to go out of the phone app
3. BLE transport however consumes ~110KB memory at runtime. If the product does not use the BLE or BT functionality after provisioning is done, almost all the memory can be reclaimed back and can be added into the heap.
4. SoftAP based transport is highly interoperable; however as the same radio is shared between SoftAP and Station interface, the transport is not reliable in the phase when the Wi-Fi connection to external AP is attempted. Also, the client may roam back to different network when the SoftAP changes the channel at the time of Station connection.
5. SoftAP transport does not require much additional memory for the Wi-Fi use-cases
6. SoftAP based provisioning requires the phone app user to go to "System Settings" to connect to Wi-Fi network hosted by the device in case of iOS. The discovery (scanning) as well as connection API is not available for the iOS applications.

Deciding on Security

Depending on the transport and other constraints the security scheme needs to be selected by the application developers. Following considerations need to be given from the provisioning security perspective: 1. The configuration data sent from the client to the device and the response has to be secured. 2. The client should authenticate the device it is connected to. 3. The device manufacturer may choose proof-of-possession - a unique per device secret to be entered on the provisioning client as a security measure to make sure that the user can provision the device in the possession.

There are two levels of security schemes. The developer may select one or combination depending on requirements.

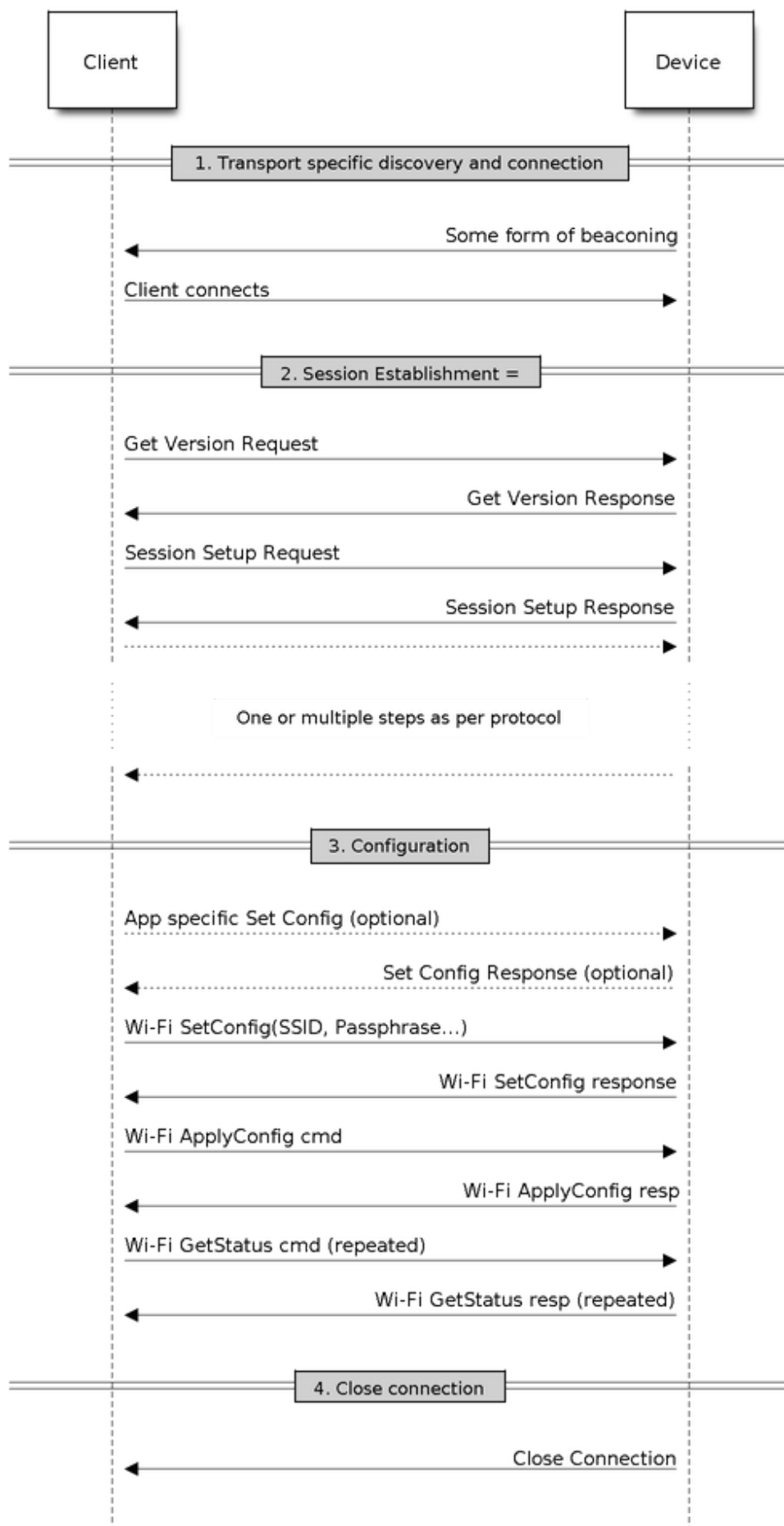


Fig. 23: Typical Provisioning Process

1. *Transport Security*: SoftAP provisioning may choose WPA2 protected security with unique per-device passphrase. Per-device unique passphrase can also act as a proof-of-possession. For BLE, “just-works” security can be used as a transport level security after understanding the level of security it provides.
2. *Application Security*: The unified provisioning subsystem provides application level security (*security1*) that provides data protection and authentication (through proof-of-possession) if the application does not use the transport level security or if the transport level security is not sufficient for the use-case.

Device Discovery

The advertisement and device discovery is left to the application and depending on the protocol chosen, the phone apps and device firmware application can choose appropriate method to advertise and discovery.

For the SoftAP+HTTP transport, typically the SSID (network name) of the AP hosted by the device can be used for discovery.

For the BLE transport device name or primary service included in the advertisement or combination of both can be used for discovery.

Architecture

The below diagram shows architecture of unified provisioning.

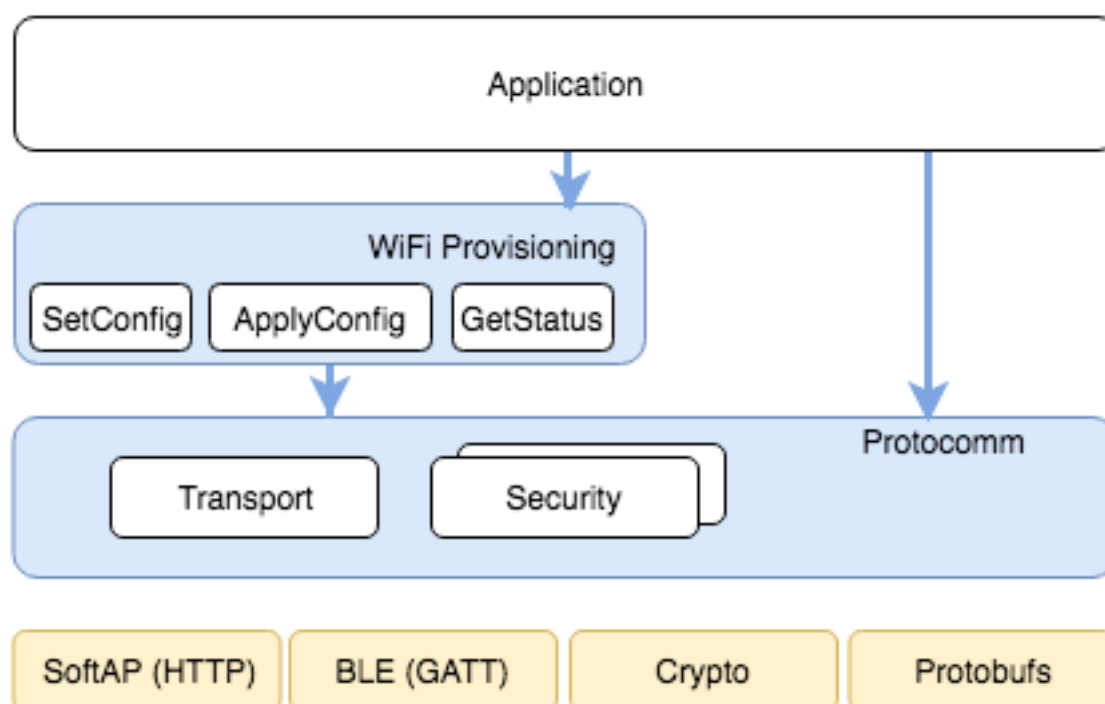


Fig. 24: Unified Provisioning Architecture

It relies on the base layer called *Protocol Communication* (Protocol Communication) which provides a framework for security schemes and transport mechanisms. Wi-Fi Provisioning layer uses Protocomm to provide simple callbacks to the application for setting the configuration and getting the Wi-Fi status. The application has control over implementation of these callbacks. In addition application can directly use protocomm to register custom handlers.

Application creates a protocomm instance which is mapped to a specific transport and specific security scheme. Each transport in the protocomm has a concept of an “end-point” which corresponds to logical channel for communication for specific type of information. For example security handshake happens on a different endpoint than the Wi-Fi

configuration endpoint. Each end-point is identified using a string and depending on the transport internal representation of the end-point changes. In case of SoftAP+HTTP transport the end-point corresponds to URI whereas in case of BLE the end-point corresponds to GATT characteristic with specific UUID. Developers can create custom end-points and implement handler for the data that is received or sent over the same end-point.

Security Schemes

At present unified provisioning supports two security schemes: 1. Security0 - No security (No encryption) 2. Security1 - Curve25519 based key exchange, shared key derivation and AES256-CTR mode encryption of the data. It supports two modes :

- a. Authorized - Proof of Possession (PoP) string used to authorize session and derive shared key
- b. No Auth (Null PoP) - Shared key derived through key exchange only

Security1 scheme details are shown in the below sequence diagram

Sample Code

Please refer to [Protocol Communication](#) and [Wi-Fi Provisioning](#) for API guides and code snippets on example usage.

Various use case implementations can be found as examples under [provisioning](#).

Provisioning Tools

Provisioning applications are available for various platforms, along with source code:

- Android : [esp-idf-provisioning-android](#)
- iOS : [esp-idf-provisioning-ios](#)
- Linux/MacOS/Windows : [tools/esp_prov](#) (a python based command line tool for provisioning)

The phone applications offer simple UI and thus more user centric, while the command line application is useful as a debugging tool for developers.

2.5.3 Wi-Fi Provisioning

Overview

This component provides APIs that control Wi-Fi provisioning service for receiving and configuring Wi-Fi credentials over SoftAP or BLE transport via secure [Protocol Communication \(protocomm\)](#) sessions. The set of `wifi_prov_mgr_` APIs help in quickly implementing a provisioning service having necessary features with minimal amount of code and sufficient flexibility.

Initialization `wifi_prov_mgr_init()` is called to configure and initialize the provisioning manager and thus this must be called prior to invoking any other `wifi_prov_mgr_` APIs. Note that the manager relies on other components of IDF, namely NVS, TCP/IP, Event Loop and Wi-Fi (and optionally mDNS), hence these must be initialized beforehand. The manager can be de-initialized at any moment by making a call to `wifi_prov_mgr_deinit()`.

```
wifi_prov_mgr_config_t config = {
    .scheme = wifi_prov_scheme_ble,
    .scheme_event_handler = WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BTDM
};

ESP_ERR_CHECK( wifi_prov_mgr_init(config) );
```

The configuration structure `wifi_prov_mgr_config_t` has a few fields to specify the behavior desired of the manager :

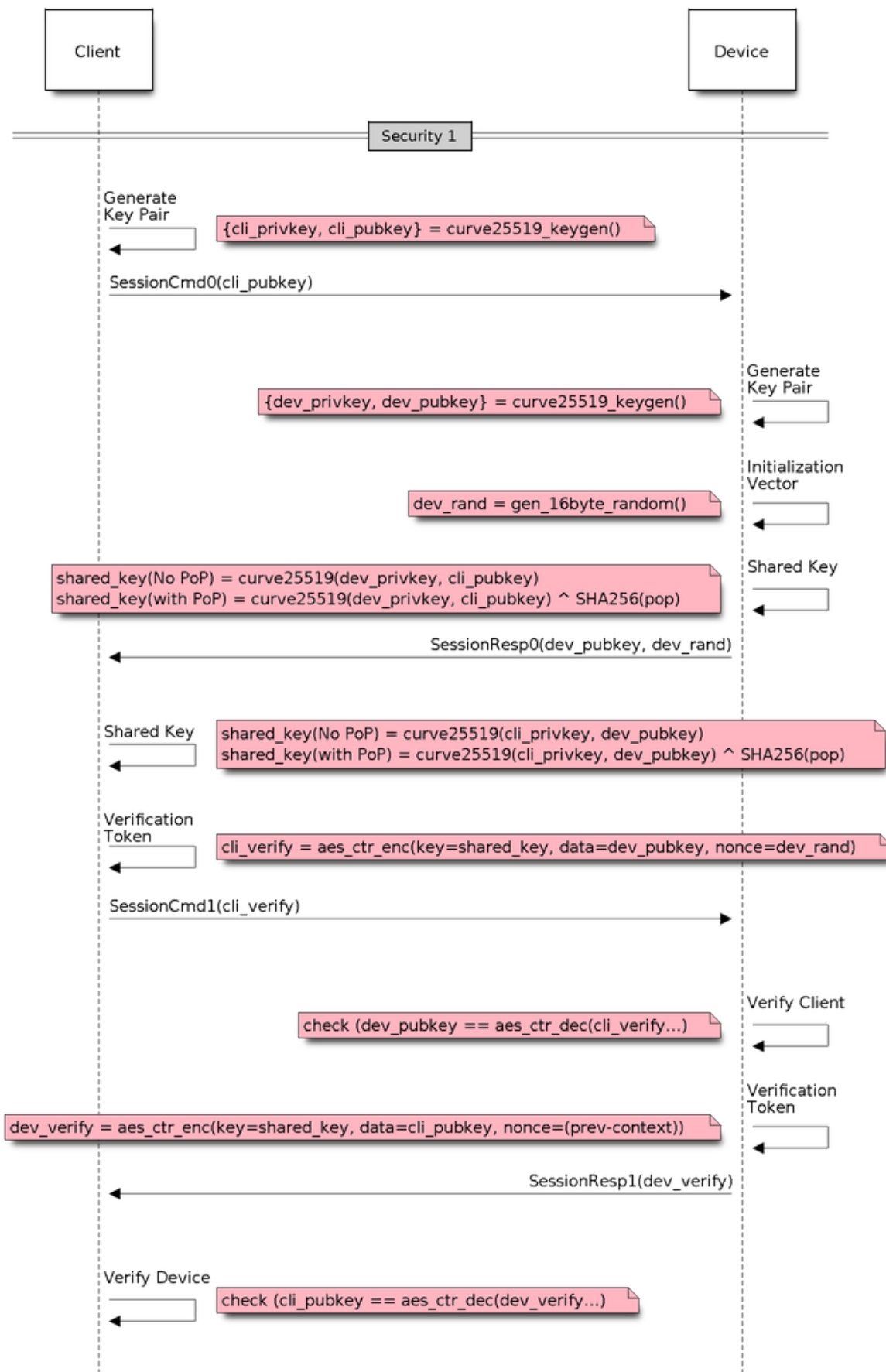


Fig. 25: Security1

- *scheme* : This is used to specify the provisioning scheme. Each scheme corresponds to one of the modes of transport supported by protocomm. Hence, we have three options :
 - `wifi_prov_scheme_ble` : BLE transport and GATT Server for handling provisioning commands
 - `wifi_prov_scheme_softap` : Wi-Fi SoftAP transport and HTTP Server for handling provisioning commands
 - `wifi_prov_scheme_console` : Serial transport and console for handling provisioning commands
- *scheme_event_handler* : An event handler defined along with scheme. Choosing appropriate scheme specific event handler allows the manager to take care of certain matters automatically. Presently this is not used for either SoftAP or Console based provisioning, but is very convenient for BLE. To understand how, we must recall that Bluetooth requires quite some amount of memory to function and once provisioning is finished, the main application may want to reclaim back this memory (or part of it, if it needs to use either BLE or classic BT). Also, upon every future reboot of a provisioned device, this reclamation of memory needs to be performed again. To reduce this complication in using `wifi_prov_scheme_ble`, the scheme specific handlers have been defined, and depending upon the chosen handler, the BLE / classic BT / BTDM memory will be freed automatically when the provisioning manager is de-initialized. The available options are:
 - `WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BTDM` - Free both classic BT and BLE (BTDM) memory. Used when main application doesn't require Bluetooth at all.
 - `WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BLE` - Free only BLE memory. Used when main application requires classic BT.
 - `WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BT` - Free only classic BT. Used when main application requires BLE. In this case freeing happens right when the manager is initialized.
 - `WIFI_PROV_EVENT_HANDLER_NONE` - Don't use any scheme specific handler. Used when provisioning scheme is not BLE (i.e. SoftAP or Console), or when main application wants to handle the memory reclaiming on its own, or needs both BLE and classic BT to function.
- *app_event_handler* (Deprecated) : It is now recommended to catch `WIFI_PROV_EVENT`s` that are emitted to the default event loop handler. See definition of `wifi_prov_cb_event_t` for the list of events that are generated by the provisioning service. Here is an excerpt showing some of the provisioning events:

```
static void event_handler(void* arg, esp_event_base_t event_base,
                        int event_id, void* event_data)
{
    if (event_base == WIFI_PROV_EVENT) {
        switch (event_id) {
            case WIFI_PROV_START:
                ESP_LOGI(TAG, "Provisioning started");
                break;
            case WIFI_PROV_CRED_RECV: {
                wifi_sta_config_t *wifi_sta_cfg = (wifi_sta_config_t_
↳*)event_data;
                ESP_LOGI(TAG, "Received Wi-Fi credentials"
                "\n\tSSID      : %s\n\tPassword : %s",
                (const char *) wifi_sta_cfg->ssid,
                (const char *) wifi_sta_cfg->password);
                break;
            }
            case WIFI_PROV_CRED_FAIL: {
                wifi_prov_sta_fail_reason_t *reason = (wifi_prov_sta_fail_
↳reason_t *)event_data;
                ESP_LOGE(TAG, "Provisioning failed!\n\tReason : %s"
                "\n\tPlease reset to factory and retry_
↳provisioning",
                (*reason == WIFI_PROV_STA_AUTH_ERROR) ?
                "Wi-Fi station authentication failed" : "Wi-Fi_
↳access-point not found");
            }
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

        break;
    }
    case WIFI_PROV_CRED_SUCCESS:
        ESP_LOGI(TAG, "Provisioning successful");
        break;
    case WIFI_PROV_END:
        /* De-initialize manager once provisioning is finished */
        wifi_prov_mgr_deinit();
        break;
    default:
        break;
}
}
}

```

The manager can be de-initialized at any moment by making a call to `wifi_prov_mgr_deinit()`.

Check Provisioning State Whether device is provisioned or not can be checked at runtime by calling `wifi_prov_mgr_is_provisioned()`. This internally checks if the Wi-Fi credentials are stored in NVS.

Note that presently manager does not have its own NVS namespace for storage of Wi-Fi credentials, instead it relies on the `esp_wifi_` APIs to set and get the credentials stored in NVS from the default location.

If provisioning state needs to be reset, any of the following approaches may be taken :

- the associated part of NVS partition has to be erased manually
- main application must implement some logic to call `esp_wifi_` APIs for erasing the credentials at runtime
- main application must implement some logic to force start the provisioning irrespective of the provisioning state

```

bool provisioned = false;
ESP_ERR_CHECK( wifi_prov_mgr_is_provisioned(&provisioned) );

```

Start Provisioning Service At the time of starting provisioning we need to specify a service name and the corresponding key. These translate to :

- Wi-Fi SoftAP SSID and passphrase, respectively, when scheme is `wifi_prov_scheme_softap`
- BLE Device name (service key is ignored) when scheme is `wifi_prov_scheme_ble`

Also, since internally the manager uses `protocomm`, we have the option of choosing one of the security features provided by it :

- Security 1 is secure communication which consists of a prior handshake involving X25519 key exchange along with authentication using a proof of possession (*pop*), followed by AES-CTR for encryption/decryption of subsequent messages
- Security 0 is simply plain text communication. In this case the *pop* is simply ignored

See [Provisioning](#) for details about the security features.

```

const char *service_name = "my_device";
const char *service_key = "password";

wifi_prov_security_t security = WIFI_PROV_SECURITY_1;
const char *pop = "abcd1234";

ESP_ERR_CHECK( wifi_prov_mgr_start_provisioning(security, pop, service_
↪name, service_key) );

```


The provisioning service will automatically finish only if it receives valid Wi-Fi AP credentials followed by successful connection of device to the AP (IP obtained). Regardless of that, the provisioning service can be stopped at any moment by making a call to `wifi_prov_mgr_stop_provisioning()`.

Note: If the device fails to connect with the provided credentials, it won't accept new credentials anymore, but the provisioning service will keep on running (only to convey failure to the client), until the device is restarted. Upon restart the provisioning state will turn out to be true this time (as credentials will be found in NVS), but device will again fail to connect with those same credentials (unless an AP with the matching credentials somehow does become available). This situation can be fixed by resetting the credentials in NVS or force starting the provisioning service. This has been explained above in [Check Provisioning State](#).

Waiting For Completion Typically, the main application will wait for the provisioning to finish, then de-initialize the manager to free up resources and finally start executing its own logic.

There are two ways for making this possible. The simpler way is to use a blocking call to `wifi_prov_mgr_wait()`.

```
// Start provisioning service
ESP_ERR_CHECK( wifi_prov_mgr_start_provisioning( security, pop, service_
↳ name, service_key ) );

// Wait for service to complete
wifi_prov_mgr_wait();

// Finally de-initialize the manager
wifi_prov_mgr_deinit();
```

The other way is to use the default event loop handler to catch `WIFI_PROV_EVENT`s` and call `:cpp:func:`wifi_prov_mgr_deinit()`` when event ID is `WIFI_PROV_END`:

```
static void event_handler(void* arg, esp_event_base_t event_base,
                          int event_id, void* event_data)
{
    if (event_base == WIFI_PROV_EVENT && event_id == WIFI_PROV_END) {
        /* De-initialize manager once provisioning is finished */
        wifi_prov_mgr_deinit();
    }
}
```

User Side Implementation When the service is started, the device to be provisioned is identified by the advertised service name which, depending upon the selected transport, is either the BLE device name or the SoftAP SSID.

When using SoftAP transport, for allowing service discovery, mDNS must be initialized before starting provisioning. In this case the hostname set by the main application is used, and the service type is internally set to `_esp_wifi_prov`.

When using BLE transport, a custom 128 bit UUID should be set using `wifi_prov_scheme_ble_set_service_uuid()`. This UUID will be included in the BLE advertisement and will correspond to the primary GATT service that provides provisioning endpoints as GATT characteristics. Each GATT characteristic will be formed using the primary service UUID as base, with different auto assigned 12th and 13th bytes (assume counting starts from 0th byte). Since, an endpoint characteristic UUID is auto assigned, it shouldn't be used to identify the endpoint. Instead, client side applications should identify the endpoints by reading the User Characteristic Description (0x2901) descriptor for each characteristic, which contains the endpoint name of the characteristic. For example, if the service UUID is set to `55cc035e-fb27-4f80-be02-3c60828b7451`, each endpoint characteristic will be assigned a UUID like `55cc____-fb27-4f80-be02-3c60828b7451`, with unique values at the 12th and 13th bytes.

Once connected to the device, the provisioning related protocomm endpoints can be identified as follows :

Table 5: Endpoints provided by Provisioning Service

| Endpoint Name (BLE + GATT Server) | URI (SoftAP + HTTP Server + mDNS) | Description |
|-----------------------------------|---|--|
| prov-session | <a href="http://<mdns-hostname>.local/prov-session">http://<mdns-hostname>.local/prov-session | Security endpoint used for session establishment |
| prov-scan | http://wifi-prov.local/prov-scan | Endpoint used for starting Wi-Fi scan and receiving scan results |
| prov-config | <a href="http://<mdns-hostname>.local/prov-config">http://<mdns-hostname>.local/prov-config | Endpoint used for configuring Wi-Fi credentials on device |
| proto-ver | <a href="http://<mdns-hostname>.local/proto-ver">http://<mdns-hostname>.local/proto-ver | Endpoint for retrieving version info |

Immediately after connecting, the client application may fetch the version / capabilities information from the *proto-ver* endpoint. All communications to this endpoint are un-encrypted, hence necessary information (that may be relevant for deciding compatibility) can be retrieved before establishing a secure session. The response is in JSON format and looks like: `prov: { ver: v1.1, cap: [no_pop] }, my_app: { ver: 1.345, cap: [cloud, local_ctrl] }, ...`. Here label *prov* provides provisioning service version (*ver*) and capabilities (*cap*). For now, only *no_pop* capability is supported, which indicates that the service doesn't require proof of possession for authentication. Any application related version / capabilities will be given by other labels (like *my_app* in this example). These additional fields are set using `wifi_prov_mgr_set_app_info()`.

User side applications need to implement the signature handshaking required for establishing and authenticating secure protocomm sessions as per the security scheme configured for use (this is not needed when manager is configured to use protocomm security 0).

See Unified Provisioning for more details about the secure handshake and encryption used. Applications must use the *.proto* files found under `protocomm/proto`, which define the Protobuf message structures supported by *prov-session* endpoint.

Once a session is established, Wi-Fi credentials are configured using the following set of *wifi_config* commands, serialized as Protobuf messages (the corresponding *.proto* files can be found under `wifi_provisioning/proto`):

- *get_status* - For querying the Wi-Fi connection status. The device will respond with a status which will be one of connecting / connected / disconnected. If status is disconnected, a disconnection reason will also be included in the status response.
- *set_config* - For setting the Wi-Fi connection credentials
- *apply_config* - For applying the credentials saved during *set_config* and start the Wi-Fi station

After session establishment, client can also request Wi-Fi scan results from the device. The results returned is a list of AP SSIDs, sorted in descending order of signal strength. This allows client applications to display APs nearby to the device at the time of provisioning, and users can select one of the SSIDs and provide the password which is then sent using the *wifi_config* commands described above. The *wifi_scan* endpoint supports the following protobuf commands:

- *scan_start* - For starting Wi-Fi scan with various options:
 - *blocking* (input) - If true, the command returns only when the scanning is finished
 - *passive* (input) - If true scan is started in passive mode (this may be slower) instead of active mode
 - *group_channels* (input) - This specifies whether to scan all channels in one go (when zero) or perform scanning of channels in groups, with 120ms delay between scanning of consecutive groups, and the value of this parameter sets the number of channels in each group. This is useful when transport mode is SoftAP, where scanning all channels in one go may not give the Wi-Fi driver enough time to send out beacons, and hence may cause disconnection with any connected stations. When scanning in groups, the manager will wait for atleast 120ms after completing scan on a group of channels, and thus allow the driver to send out the beacons. For example, given that the total number of Wi-Fi channels is 14, then setting *group_channels* to 4, will create 5 groups, with each group having 3 channels, except the last one which will have $14 \% 3 = 2$ channels. So, when scan is started, the first 3 channels will be scanned, followed by a 120ms delay, and then the next 3 channels, and so on, until all the 14 channels have been

scanned. One may need to adjust this parameter as having only few channels in a group may slow down the overall scan time, while having too many may again cause disconnection. Usually a value of 4 should work for most cases. Note that for any other mode of transport, e.g. BLE, this can be safely set to 0, and hence achieve the fastest overall scanning time.

- *period_ms* (input) - Scan parameter specifying how long to wait on each channel
- *scan_status* - Gives the status of scanning process :
 - *scan_finished* (output) - When scan has finished this returns true
 - *result_count* (output) - This gives the total number of results obtained till now. If scan is yet happening this number will keep on updating
- *scan_result* - For fetching scan results. This can be called even if scan is still on going
 - *start_index* (input) - Starting index from where to fetch the entries from the results list
 - *count* (input) - Number of entries to fetch from the starting index
 - *entries* (output) - List of entries returned. Each entry consists of *ssid*, *channel* and *rss* information

Additional Endpoints In case users want to have some additional protocomm endpoints customized to their requirements, this is done in two steps. First is creation of an endpoint with a specific name, and the second step is the registration of a handler for this endpoint. See [protocomm](#) for the function signature of an endpoint handler. A custom endpoint must be created after initialization and before starting the provisioning service. Whereas, the protocomm handler is registered for this endpoint only after starting the provisioning service.

```
wifi_prov_mgr_init(config);
wifi_prov_mgr_endpoint_create("custom-endpoint");
wifi_prov_mgr_start_provisioning(security, pop, service_name, service_
↪key);
wifi_prov_mgr_endpoint_register("custom-endpoint", custom_ep_handler, ↪
↪custom_ep_data);
```

When the provisioning service stops, the endpoint is unregistered automatically.

One can also choose to call [wifi_prov_mgr_endpoint_unregister\(\)](#) to manually deactivate an endpoint at runtime. This can also be used to deactivate the internal endpoints used by the provisioning service.

When / How To Stop Provisioning Service? The default behavior is that once the device successfully connects using the Wi-Fi credentials set by the [apply_config](#) command, the provisioning service will be stopped (and BLE / SoftAP turned off) automatically after responding to the next [get_status](#) command. If [get_status](#) command is not received by the device, the service will be stopped after a 30s timeout.

On the other hand, if device was not able to connect using the provided Wi-Fi credentials, due to incorrect SSID / passphrase, the service will keep running, and [get_status](#) will keep responding with disconnected status and reason for disconnection. Any further attempts to provide another set of Wi-Fi credentials, will be rejected. These credentials will be preserved, unless the provisioning service is force started, or NVS erased.

If this default behavior is not desired, it can be disabled by calling [wifi_prov_mgr_disable_auto_stop\(\)](#). Now the provisioning service will only be stopped after an explicit call to [wifi_prov_mgr_stop_provisioning\(\)](#), which returns immediately after scheduling a task for stopping the service. The service stops after a certain delay and WIFI_PROV_END event gets emitted. This delay is specified by the argument to [wifi_prov_mgr_disable_auto_stop\(\)](#).

The customized behavior is useful for applications which want the provisioning service to be stopped some time after the Wi-Fi connection is successfully established. For example, if the application requires the device to connect to some cloud service and obtain another set of credentials, and exchange this credentials over a custom protocomm endpoint, then after successfully doing so stop the provisioning service by calling [wifi_prov_mgr_stop_provisioning\(\)](#) inside the protocomm handler itself. The right amount of delay ensures that the transport resources are freed only after the response from the protocomm handler reaches the client side application.

Application Examples

For complete example implementation see [provisioning/manager](#)

API Reference

Header File

- [wifi_provisioning/include/wifi_provisioning/manager.h](#)

Functions

esp_err_t **wifi_prov_mgr_init** (*wifi_prov_mgr_config_t* config)

Initialize provisioning manager instance.

Configures the manager and allocates internal resources

Configuration specifies the provisioning scheme (transport) and event handlers

Event WIFI_PROV_INIT is emitted right after initialization is complete

Return

- ESP_OK : Success
- ESP_FAIL : Fail

Parameters

- [in] config: Configuration structure

void **wifi_prov_mgr_deinit** (void)

Stop provisioning (if running) and release resource used by the manager.

Event WIFI_PROV_DEINIT is emitted right after de-initialization is finished

If provisioning service is still active when this API is called, it first stops the service, hence emitting WIFI_PROV_END, and then performs the de-initialization

esp_err_t **wifi_prov_mgr_is_provisioned** (bool *provisioned)

Checks if device is provisioned.

This checks if Wi-Fi credentials are present on the NVS

The Wi-Fi credentials are assumed to be kept in the same NVS namespace as used by esp_wifi component

If one were to call esp_wifi_set_config() directly instead of going through the provisioning process, this function will still yield true (i.e. device will be found to be provisioned)

Note Calling wifi_prov_mgr_start_provisioning() automatically resets the provision state, irrespective of what the state was prior to making the call.

Return

- ESP_OK : Retrieved provision state successfully
- ESP_FAIL : Wi-Fi not initialized
- ESP_ERR_INVALID_ARG : Null argument supplied
- ESP_ERR_INVALID_STATE : Manager not initialized

Parameters

- [out] provisioned: True if provisioned, else false

esp_err_t **wifi_prov_mgr_start_provisioning** (*wifi_prov_security_t* security, const char *pop, const char *service_name, const char *service_key)

Start provisioning service.

This starts the provisioning service according to the scheme configured at the time of initialization. For scheme :

- wifi_prov_scheme_ble : This starts protocomm_ble, which internally initializes BLE transport and starts GATT server for handling provisioning requests
- wifi_prov_scheme_softap : This activates SoftAP mode of Wi-Fi and starts protocomm_httpd, which internally starts an HTTP server for handling provisioning requests (If mDNS is active it also starts advertising service with type _esp_wifi_prov._tcp)

Event WIFI_PROV_START is emitted right after provisioning starts without failure

Note This API will start provisioning service even if device is found to be already provisioned, i.e. `wifi_prov_mgr_is_provisioned()` yields true

Return

- `ESP_OK` : Provisioning started successfully
- `ESP_FAIL` : Failed to start provisioning service
- `ESP_ERR_INVALID_STATE` : Provisioning manager not initialized or already started

Parameters

- `[in] security`: Specify which protocomm security scheme to use :
 - `WIFI_PROV_SECURITY_0` : For no security
 - `WIFI_PROV_SECURITY_1` : x25519 secure handshake for session establishment followed by AES-CTR encryption of provisioning messages
- `[in] pop`: Pointer to proof of possession string (NULL if not needed). This is relevant only for protocomm security 1, in which case it is used for authenticating secure session
- `[in] service_name`: Unique name of the service. This translates to:
 - Wi-Fi SSID when provisioning mode is softAP
 - Device name when provisioning mode is BLE
- `[in] service_key`: Key required by client to access the service (NULL if not needed). This translates to:
 - Wi-Fi password when provisioning mode is softAP
 - ignored when provisioning mode is BLE

void `wifi_prov_mgr_stop_provisioning` (void)

Stop provisioning service.

If provisioning service is active, this API will initiate a process to stop the service and return. Once the service actually stops, the event `WIFI_PROV_END` will be emitted.

If `wifi_prov_mgr_deinit()` is called without calling this API first, it will automatically stop the provisioning service and emit the `WIFI_PROV_END`, followed by `WIFI_PROV_DEINIT`, before returning.

This API will generally be used along with `wifi_prov_mgr_disable_auto_stop()` in the scenario when the main application has registered its own endpoints, and wishes that the provisioning service is stopped only when some protocomm command from the client side application is received.

Calling this API inside an endpoint handler, with sufficient `cleanup_delay`, will allow the response / acknowledgment to be sent successfully before the underlying protocomm service is stopped.

`Cleaup_delay` is set when calling `wifi_prov_mgr_disable_auto_stop()`. If not specified, it defaults to 1000ms.

For straightforward cases, using this API is usually not necessary as provisioning is stopped automatically once `WIFI_PROV_CRED_SUCCESS` is emitted. Stopping is delayed (maximum 30 seconds) thus allowing the client side application to query for Wi-Fi state, i.e. after receiving the first query and sending `Wi-Fi state connected` response the service is stopped immediately.

void `wifi_prov_mgr_wait` (void)

Wait for provisioning service to finish.

Calling this API will block until provisioning service is stopped i.e. till event `WIFI_PROV_END` is emitted.

This will not block if provisioning is not started or not initialized.

esp_err_t `wifi_prov_mgr_disable_auto_stop` (*uint32_t cleanup_delay*)

Disable auto stopping of provisioning service upon completion.

By default, once provisioning is complete, the provisioning service is automatically stopped, and all endpoints (along with those registered by main application) are deactivated.

This API is useful in the case when main application wishes to close provisioning service only after it receives some protocomm command from the client side app. For example, after connecting to Wi-Fi, the device may want to connect to the cloud, and only once that is successfully, the device is said to be fully configured. But, then it is upto the main application to explicitly call `wifi_prov_mgr_stop_provisioning()` later when the device is fully configured and the provisioning service is no longer required.

Note This must be called before executing `wifi_prov_mgr_start_provisioning()`

Return

- ESP_OK : Success
- ESP_ERR_INVALID_STATE : Manager not initialized or provisioning service already started

Parameters

- [in] `cleanup_delay`: Sets the delay after which the actual cleanup of transport related resources is done after a call to `wifi_prov_mgr_stop_provisioning()` returns. Minimum allowed value is 100ms. If not specified, this will default to 1000ms.

esp_err_t `wifi_prov_mgr_set_app_info` (`const` char **label*, `const` char **version*, `const` char ***capabilities*, `size_t` *total_capabilities*)

Set application version and capabilities in the JSON data returned by proto-ver endpoint.

This function can be called multiple times, to specify information about the various application specific services running on the device, identified by unique labels.

The provisioning service itself registers an entry in the JSON data, by the label “prov” , containing only provisioning service version and capabilities. Application services should use a label other than “prov” so as not to overwrite this.

Note This must be called before executing `wifi_prov_mgr_start_provisioning()`

Return

- ESP_OK : Success
- ESP_ERR_INVALID_STATE : Manager not initialized or provisioning service already started
- ESP_ERR_NO_MEM : Failed to allocate memory for version string
- ESP_ERR_INVALID_ARG : Null argument

Parameters

- [in] `label`: String indicating the application name.
- [in] `version`: String indicating the application version. There is no constraint on format.
- [in] `capabilities`: Array of strings with capabilities. These could be used by the client side app to know the application registered endpoint capabilities
- [in] `total_capabilities`: Size of capabilities array

esp_err_t `wifi_prov_mgr_endpoint_create` (`const` char **ep_name*)

Create an additional endpoint and allocate internal resources for it.

This API is to be called by the application if it wants to create an additional endpoint. All additional endpoints will be assigned UUIDs starting from 0xFF54 and so on in the order of execution.

protocomm handler for the created endpoint is to be registered later using `wifi_prov_mgr_endpoint_register()` after provisioning has started.

Note This API can only be called BEFORE provisioning is started

Note Additional endpoints can be used for configuring client provided parameters other than Wi-Fi credentials, that are necessary for the main application and hence must be set prior to starting the application

Note After session establishment, the additional endpoints must be targeted first by the client side application before sending Wi-Fi configuration, because once Wi-Fi configuration finishes the provisioning service is stopped and hence all endpoints are unregistered

Return

- ESP_OK : Success
- ESP_FAIL : Failure

Parameters

- [in] `ep_name`: unique name of the endpoint

esp_err_t `wifi_prov_mgr_endpoint_register` (`const` char **ep_name*, *proto-comm_req_handler_t* *handler*, void **user_ctx*)

Register a handler for the previously created endpoint.

This API can be called by the application to register a protocomm handler to any endpoint that was created using `wifi_prov_mgr_endpoint_create()`.

Note This API can only be called AFTER provisioning has started

Note Additional endpoints can be used for configuring client provided parameters other than Wi-Fi credentials, that are necessary for the main application and hence must be set prior to starting the application

Note After session establishment, the additional endpoints must be targeted first by the client side application before sending Wi-Fi configuration, because once Wi-Fi configuration finishes the provisioning service

is stopped and hence all endpoints are unregistered

Return

- ESP_OK : Success
- ESP_FAIL : Failure

Parameters

- [in] ep_name: Name of the endpoint
- [in] handler: Endpoint handler function
- [in] user_ctx: User data

void **wifi_prov_mgr_endpoint_unregister** (const char *ep_name)

Unregister the handler for an endpoint.

This API can be called if the application wants to selectively unregister the handler of an endpoint while the provisioning is still in progress.

All the endpoint handlers are unregistered automatically when the provisioning stops.

Parameters

- [in] ep_name: Name of the endpoint

esp_err_t **wifi_prov_mgr_event_handler** (void *ctx, *system_event_t* *event)

Event handler for provisioning manager.

This is called from the main event handler and controls the provisioning manager's internal state machine depending on incoming Wi-Fi events

Note : This function is DEPRECATED, because events are now handled internally using the event loop library, esp_event. Calling this will do nothing and simply return ESP_OK.

Return

- ESP_OK : Event handled successfully

Parameters

- [in] ctx: Event context data
- [in] event: Event info

esp_err_t **wifi_prov_mgr_get_wifi_state** (*wifi_prov_sta_state_t* *state)

Get state of Wi-Fi Station during provisioning.

Return

- ESP_OK : Successfully retrieved Wi-Fi state
- ESP_FAIL : Provisioning app not running

Parameters

- [out] state: Pointer to wifi_prov_sta_state_t variable to be filled

esp_err_t **wifi_prov_mgr_get_wifi_disconnect_reason** (*wifi_prov_sta_fail_reason_t* *reason)

Get reason code in case of Wi-Fi station disconnection during provisioning.

Return

- ESP_OK : Successfully retrieved Wi-Fi disconnect reason
- ESP_FAIL : Provisioning app not running

Parameters

- [out] reason: Pointer to wifi_prov_sta_fail_reason_t variable to be filled

esp_err_t **wifi_prov_mgr_configure_sta** (*wifi_config_t* *wifi_cfg)

Runs Wi-Fi as Station with the supplied configuration.

Configures the Wi-Fi station mode to connect to the AP with SSID and password specified in config structure and sets Wi-Fi to run as station.

This is automatically called by provisioning service upon receiving new credentials.

If credentials are to be supplied to the manager via a different mode other than through protocomm, then this API needs to be called.

Event WIFI_PROV_CRED_RECV is emitted after credentials have been applied and Wi-Fi station started

Return

- ESP_OK : Wi-Fi configured and started successfully
- ESP_FAIL : Failed to set configuration

Parameters

- [in] `wifi_cfg`: Pointer to Wi-Fi configuration structure

Structures**struct `wifi_prov_event_handler_t`**

Event handler that is used by the manager while provisioning service is active.

Public Members*wifi_prov_cb_func_t* **event_cb**

Callback function to be executed on provisioning events

void ***user_data**

User context data to pass as parameter to callback function

struct `wifi_prov_scheme`

Structure for specifying the provisioning scheme to be followed by the manager.

Note Ready to use schemes are available:

- `wifi_prov_scheme_ble` : for provisioning over BLE transport + GATT server
- `wifi_prov_scheme_softap` : for provisioning over SoftAP transport + HTTP server
- `wifi_prov_scheme_console` : for provisioning over Serial UART transport + Console (for debugging)

Public Members*esp_err_t* (***prov_start**) (*protocomm_t* *pc, void *config)

Function which is to be called by the manager when it is to start the provisioning service associated with a protocomm instance and a scheme specific configuration

esp_err_t (***prov_stop**) (*protocomm_t* *pc)

Function which is to be called by the manager to stop the provisioning service previously associated with a protocomm instance

void **(*new_config)** (void)Function which is to be called by the manager to generate a new configuration for the provisioning service, that is to be passed to *prov_start()*void **(*delete_config)** (void *config)Function which is to be called by the manager to delete a configuration generated using *new_config()**esp_err_t* (***set_config_service**) (void *config, **const** char *service_name, **const** char *service_key)

Function which is to be called by the manager to set the service name and key values in the configuration structure

esp_err_t (***set_config_endpoint**) (void *config, **const** char *endpoint_name, uint16_t uuid)

Function which is to be called by the manager to set a protocomm endpoint with an identifying name and UUID in the configuration structure

wifi_mode_t **wifi_mode**

Sets mode of operation of Wi-Fi during provisioning This is set to :

- `WIFI_MODE_APSTA` for SoftAP transport
- `WIFI_MODE_STA` for BLE transport

struct `wifi_prov_mgr_config_t`

Structure for specifying the manager configuration.

Public Members

wifi_prov_scheme_t **scheme**

Provisioning scheme to use. Following schemes are already available:

- `wifi_prov_scheme_ble` : for provisioning over BLE transport + GATT server
- `wifi_prov_scheme_softap` : for provisioning over SoftAP transport + HTTP server + mDNS (optional)
- `wifi_prov_scheme_console` : for provisioning over Serial UART transport + Console (for debugging)

wifi_prov_event_handler_t **scheme_event_handler**

Event handler required by the scheme for incorporating scheme specific behavior while provisioning manager is running. Various options may be provided by the scheme for setting this field. Use `WIFI_PROV_EVENT_HANDLER_NONE` when not used. When using scheme `wifi_prov_scheme_ble`, the following options are available:

- `WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BTDM`
- `WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BLE`
- `WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BT`

wifi_prov_event_handler_t **app_event_handler**

Event handler that can be set for the purpose of incorporating application specific behavior. Use `WIFI_PROV_EVENT_HANDLER_NONE` when not used.

Macros

WIFI_PROV_EVENT_HANDLER_NONE

Event handler can be set to none if not used.

Type Definitions

```
typedef void (*wifi_prov_cb_func_t) (void *user_data, wifi_prov_cb_event_t event, void *event_data)
```

```
typedef struct wifi_prov_scheme wifi_prov_scheme_t
```

Structure for specifying the provisioning scheme to be followed by the manager.

Note Ready to use schemes are available:

- `wifi_prov_scheme_ble` : for provisioning over BLE transport + GATT server
- `wifi_prov_scheme_softap` : for provisioning over SoftAP transport + HTTP server
- `wifi_prov_scheme_console` : for provisioning over Serial UART transport + Console (for debugging)

```
typedef enum wifi_prov_security wifi_prov_security_t
```

Security modes supported by the Provisioning Manager.

These are same as the security modes provided by protocomm

Enumerations

```
enum wifi_prov_cb_event_t
```

Events generated by manager.

These events are generated in order of declaration and, for the stretch of time between initialization and de-initialization of the manager, each event is signaled only once

Values:

```
WIFI_PROV_INIT
```

Emitted when the manager is initialized

```
WIFI_PROV_START
```

Indicates that provisioning has started

```
WIFI_PROV_CRED_RECV
```

Emitted when Wi-Fi AP credentials are received via protocomm endpoint `wifi_config`. The event data in this case is a pointer to the corresponding `wifi_sta_config_t` structure

WIFI_PROV_CRED_FAIL

Emitted when device fails to connect to the AP of which the credentials were received earlier on event `WIFI_PROV_CRED_RECV`. The event data in this case is a pointer to the disconnection reason code with type `wifi_prov_sta_fail_reason_t`

WIFI_PROV_CRED_SUCCESS

Emitted when device successfully connects to the AP of which the credentials were received earlier on event `WIFI_PROV_CRED_RECV`

WIFI_PROV_END

Signals that provisioning service has stopped

WIFI_PROV_DEINIT

Signals that manager has been de-initialized

enum wifi_prov_security

Security modes supported by the Provisioning Manager.

These are same as the security modes provided by `protocomm`

Values:

WIFI_PROV_SECURITY_0 = 0

No security (plain-text communication)

WIFI_PROV_SECURITY_1

This secure communication mode consists of X25519 key exchange

- proof of possession (pop) based authentication
- AES-CTR encryption

Header File

- [wifi_provisioning/include/wifi_provisioning/scheme_ble.h](#)

Functions

void `wifi_prov_scheme_ble_event_cb_free_bt`(void **user_data*, *wifi_prov_cb_event_t* event, void **event_data*)

void `wifi_prov_scheme_ble_event_cb_free_ble`(void **user_data*, *wifi_prov_cb_event_t* event, void **event_data*)

void `wifi_prov_scheme_ble_event_cb_free_bt`(void **user_data*, *wifi_prov_cb_event_t* event, void **event_data*)

esp_err_t `wifi_prov_scheme_ble_set_service_uuid`(uint8_t **uuid128*)

Set the 128 bit GATT service UUID used for provisioning.

This API is used to override the default 128 bit provisioning service UUID, which is 0000ffff-0000-1000-8000-00805f9b34fb.

This must be called before starting provisioning, i.e. before making a call to `wifi_prov_mgr_start_provisioning()`, otherwise the default UUID will be used.

Note The data being pointed to by the argument must be valid atleast till provisioning is started. Upon start, the manager will store an internal copy of this UUID, and this data can be freed or invalidated afterwards.

Return

- `ESP_OK` : Success
- `ESP_ERR_INVALID_ARG` : Null argument

Parameters

- [in] `uuid128`: A custom 128 bit UUID

Macros

`WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BTDM`

`WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BLE`

WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BT**Header File**

- [wifi_provisioning/include/wifi_provisioning/scheme_softap.h](#)

Functions

void **wifi_prov_scheme_softap_set_httpd_handle** (void **handle*)

Provide HTTPD Server handle externally.

Useful in cases wherein applications need the webserver for some different operations, and do not want the wifi provisioning component to start/stop a new instance.

Note This API should be called before `wifi_prov_mgr_start_provisioning()`

Parameters

- [in] *handle*: Handle to HTTPD server instance

Header File

- [wifi_provisioning/include/wifi_provisioning/scheme_console.h](#)

Header File

- [wifi_provisioning/include/wifi_provisioning/wifi_config.h](#)

Functions

esp_err_t **wifi_prov_config_data_handler** (uint32_t *session_id*, const uint8_t **inbuf*, ssize_t *inlen*, uint8_t ***outbuf*, ssize_t **outlen*, void **priv_data*)

Handler for receiving and responding to requests from master.

This is to be registered as the `wifi_config` endpoint handler (protocomm `protocomm_req_handler_t`) using `protocomm_add_endpoint()`

Structures

struct wifi_prov_sta_conn_info_t

WiFi STA connected status information.

Public Members

char **ip_addr**[IP4ADDR_STRLEN_MAX]

IP Address received by station

char **bssid**[6]

BSSID of the AP to which connection was established

char **ssid**[33]

SSID of the to which connection was established

uint8_t **channel**

Channel of the AP

uint8_t **auth_mode**

Authorization mode of the AP

struct wifi_prov_config_get_data_t

WiFi status data to be sent in response to `get_status` request from master.

Public Members

wifi_prov_sta_state_t **wifi_state**

WiFi state of the station

wifi_prov_sta_fail_reason_t **fail_reason**

Reason for disconnection (valid only when *wifi_state* is `WIFI_STATION_DISCONNECTED`)

wifi_prov_sta_conn_info_t **conn_info**

Connection information (valid only when *wifi_state* is `WIFI_STATION_CONNECTED`)

struct wifi_prov_config_set_data_t

WiFi config data received by slave during `set_config` request from master.

Public Members

char **ssid**[33]

SSID of the AP to which the slave is to be connected

char **password**[64]

Password of the AP

char **bssid**[6]

BSSID of the AP

uint8_t **channel**

Channel of the AP

struct wifi_prov_config_handlers

Internal handlers for receiving and responding to protocomm requests from master.

This is to be passed as `priv_data` for protocomm request handler (refer to `wifi_prov_config_data_handler()`) when calling `protocomm_add_endpoint()`.

Public Members

esp_err_t (***get_status_handler**) (*wifi_prov_config_get_data_t* *resp_data, *wifi_prov_ctx_t* **ctx)

Handler function called when connection status of the slave (in WiFi station mode) is requested

esp_err_t (***set_config_handler**) (**const** *wifi_prov_config_set_data_t* *req_data, *wifi_prov_ctx_t* **ctx)

Handler function called when WiFi connection configuration (eg. AP SSID, password, etc.) of the slave (in WiFi station mode) is to be set to user provided values

esp_err_t (***apply_config_handler**) (*wifi_prov_ctx_t* **ctx)

Handler function for applying the configuration that was set in `set_config_handler`. After applying the station may get connected to the AP or may fail to connect. The slave must be ready to convey the updated connection status information when `get_status_handler` is invoked again by the master.

wifi_prov_ctx_t *ctx

Context pointer to be passed to above handler functions upon invocation

Type Definitions

typedef struct *wifi_prov_ctx* **wifi_prov_ctx_t**

Type of context data passed to each get/set/apply handler function set in *wifi_prov_config_handlers* structure.

This is passed as an opaque pointer, thereby allowing it be defined later in application code as per requirements.

typedef struct *wifi_prov_config_handlers* **wifi_prov_config_handlers_t**

Internal handlers for receiving and responding to protocomm requests from master.

This is to be passed as `priv_data` for `protocomm` request handler (refer to `wifi_prov_config_data_handler()`) when calling `protocomm_add_endpoint()`.

Enumerations

`enum wifi_prov_sta_state_t`

WiFi STA status for conveying back to the provisioning master.

Values:

`WIFI_PROV_STA_CONNECTING`

`WIFI_PROV_STA_CONNECTED`

`WIFI_PROV_STA_DISCONNECTED`

`enum wifi_prov_sta_fail_reason_t`

WiFi STA connection fail reason.

Values:

`WIFI_PROV_STA_AUTH_ERROR`

`WIFI_PROV_STA_AP_NOT_FOUND`

Code examples for this API section are provided in the [provisioning](#) directory of ESP-IDF examples.

2.6 Storage API

2.6.1 FAT Filesystem Support

ESP-IDF uses the [FatFs](#) library to work with FAT filesystems. FatFs resides in the `fatfs` component. Although the library can be used directly, many of its features can be accessed via VFS, using the C standard library and POSIX API functions.

Additionally, FatFs has been modified to support the runtime pluggable disk I/O layer. This allows mapping of FatFs drives to physical disks at runtime.

Using FatFs with VFS

The header file `fatfs/vfs/esp_vfs_fat.h` defines the functions for connecting FatFs and VFS.

The function `esp_vfs_fat_register()` allocates a FATFS structure and registers a given path prefix in VFS. Subsequent operations on files starting with this prefix are forwarded to FatFs APIs. The function `esp_vfs_fat_unregister_path()` deletes the registration with VFS, and frees the FATFS structure.

Most applications use the following workflow when working with `esp_vfs_fat_` functions:

1. Call `esp_vfs_fat_register()` to specify:
 - Path prefix where to mount the filesystem (e.g. `"/sdcard"`, `"/spiflash"`)
 - FatFs drive number
 - A variable which will receive the pointer to the FATFS structure
2. Call `ff_diskio_register()` to register the disk I/O driver for the drive number used in Step 1.
3. Call the FatFs function `f_mount`, and optionally `f_fdisk`, `f_mkfs`, to mount the filesystem using the same drive number which was passed to `esp_vfs_fat_register()`. For more information, see *FatFs documentation* <<http://www.elm-chan.org/ffsw/ff/doc/mount.html>>.
4. Call the C standard library and POSIX API functions to perform such actions on files as open, read, write, erase, copy, etc. Use paths starting with the path prefix passed to `esp_vfs_register()` (for example, `"/sdcard/hello.txt"`).
5. Optionally, call the FatFs library functions directly. In this case, use paths without a VFS prefix (for example, `"/hello.txt"`).
6. Close all open files.

7. Call the FatFs function `f_mount` for the same drive number, with `NULL` `FATFS*` argument, to unmount the filesystem.
8. Call the FatFs function `ff_diskio_register()` with `NULL` `ff_diskio_impl_t*` argument and the same drive number to unregister the disk I/O driver.
9. Call `esp_vfs_fat_unregister_path()` with the path where the file system is mounted to remove FatFs from VFS, and free the `FATFS` structure allocated in Step 1.

The convenience functions `esp_vfs_fat_sdmmc_mount` and `esp_vfs_fat_sdmmc_unmount` wrap the steps described above and also handle SD card initialization. These two functions are described in the next section.

`esp_err_t esp_vfs_fat_register` (`const` char **base_path*, `const` char **fat_drive*, `size_t` *max_files*, `FATFS` ***out_fs*)

Register FATFS with VFS component.

This function registers given FAT drive in VFS, at the specified base path. If only one drive is used, `fat_drive` argument can be an empty string. Refer to FATFS library documentation on how to specify FAT drive. This function also allocates `FATFS` structure which should be used for `f_mount` call.

Note This function doesn't mount the drive into FATFS, it just connects POSIX and C standard library IO function with FATFS. You need to mount desired drive into FATFS separately.

Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_STATE` if `esp_vfs_fat_register` was already called
- `ESP_ERR_NO_MEM` if not enough memory or too many VFSes already registered

Parameters

- `base_path`: path prefix where FATFS should be registered
- `fat_drive`: FATFS drive specification; if only one drive is used, can be an empty string
- `max_files`: maximum number of files which can be open at the same time
- [`out`] `out_fs`: pointer to `FATFS` structure which can be used for FATFS `f_mount` call is returned via this argument.

`esp_err_t esp_vfs_fat_unregister_path` (`const` char **base_path*)

Un-register FATFS from VFS.

Note `FATFS` structure returned by `esp_vfs_fat_register` is destroyed after this call. Make sure to call `f_mount` function to unmount it before calling `esp_vfs_fat_unregister_ctx`. Difference between this function and the one above is that this one will release the correct drive, while the one above will release the last registered one

Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_STATE` if `FATFS` is not registered in VFS

Parameters

- `base_path`: path prefix where `FATFS` is registered. This is the same used when `esp_vfs_fat_register` was called

Using FatFs with VFS and SD cards

The header file `fatfs/vfs/esp_vfs_fat.h` defines convenience functions `esp_vfs_fat_sdmmc_mount()` and `esp_vfs_fat_sdmmc_unmount()`. These function perform Steps 1–3 and 7–9 respectively and handle SD card initialization, but provide only limited error handling. Developers are encouraged to check its source code and incorporate more advanced features into production applications.

The convenience function `esp_vfs_fat_sdmmc_unmount()` unmounts the filesystem and releases the resources acquired by `esp_vfs_fat_sdmmc_mount()`.

`esp_err_t esp_vfs_fat_sdmmc_mount` (`const` char **base_path*, `const` `sdmmc_host_t` **host_config*, `const` void **slot_config*, `const` `esp_vfs_fat_mount_config_t` **mount_config*, `sdmmc_card_t` ***out_card*)

Convenience function to get FAT filesystem on SD card registered in VFS.

This is an all-in-one function which does the following:

- initializes SDMMC driver or SPI driver with configuration in `host_config`

- initializes SD card with configuration in `slot_config`
- mounts FAT partition on SD card using FATFS library, with configuration in `mount_config`
- registers FATFS library with VFS, with prefix given by `base_prefix` variable

This function is intended to make example code more compact. For real world applications, developers should implement the logic of probing SD card, locating and mounting partition, and registering FATFS in VFS, with proper error checking and handling of exceptional conditions.

Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_STATE` if `esp_vfs_fat_sdmmc_mount` was already called
- `ESP_ERR_NO_MEM` if memory can not be allocated
- `ESP_FAIL` if partition can not be mounted
- other error codes from SDMMC or SPI drivers, SDMMC protocol, or FATFS drivers

Parameters

- `base_path`: path where partition should be registered (e.g. `"/sdcard"`)
- `host_config`: Pointer to structure describing SDMMC host. When using SDMMC peripheral, this structure can be initialized using `SDMMC_HOST_DEFAULT()` macro. When using SPI peripheral, this structure can be initialized using `SDSPI_HOST_DEFAULT()` macro.
- `slot_config`: Pointer to structure with slot configuration. For SDMMC peripheral, pass a pointer to `sdmmc_slot_config_t` structure initialized using `SDMMC_SLOT_CONFIG_DEFAULT`. For SPI peripheral, pass a pointer to `sdspi_slot_config_t` structure initialized using `SDSPI_SLOT_CONFIG_DEFAULT`.
- `mount_config`: pointer to structure with extra parameters for mounting FATFS
- `[out] out_card`: if not `NULL`, pointer to the card information structure will be returned via this argument

struct esp_vfs_fat_mount_config_t

Configuration arguments for `esp_vfs_fat_sdmmc_mount` and `esp_vfs_fat_spiflash_mount` functions.

Public Members**bool format_if_mount_failed**

If FAT partition can not be mounted, and this parameter is true, create partition table and format the filesystem.

int max_files

Max number of open files.

size_t allocation_unit_size

If `format_if_mount_failed` is set, and mount fails, format the card with given allocation unit size. Must be a power of 2, between sector size and $128 * \text{sector size}$. For SD cards, sector size is always 512 bytes. For wear_leveling, sector size is determined by `CONFIG_WL_SECTOR_SIZE` option.

Using larger allocation unit size will result in higher read/write performance and higher overhead when storing small files.

Setting this field to 0 will result in allocation unit set to the sector size.

esp_err_t esp_vfs_fat_sdmmc_unmount (void)

Unmount FAT filesystem and release resources acquired using `esp_vfs_fat_sdmmc_mount`.

Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_STATE` if `esp_vfs_fat_sdmmc_mount` hasn't been called

Using FatFs with VFS in read-only mode

The header file `fatfs/vfs/esp_vfs_fat.h` also defines the convenience functions `esp_vfs_fat_rawflash_mount()` and `esp_vfs_fat_rawflash_unmount()`. These functions perform Steps 1-3 and 7-9 respectively for read-only FAT partitions. These are particularly helpful for

data partitions written only once during factory provisioning which will not be changed by production application throughout the lifetime of the hardware.

```
esp_err_t esp_vfs_fat_rawflash_mount (const char *base_path, const char *partition_label,
                                     const esp_vfs_fat_mount_config_t *mount_config)
```

Convenience function to initialize read-only FAT filesystem and register it in VFS.

This is an all-in-one function which does the following:

- finds the partition with defined partition_label. Partition label should be configured in the partition table.
- mounts FAT partition using FATFS library
- registers FATFS library with VFS, with prefix given by base_prefix variable

Note Wear levelling is not used when FAT is mounted in read-only mode using this function.

Return

- ESP_OK on success
- ESP_ERR_NOT_FOUND if the partition table does not contain FATFS partition with given label
- ESP_ERR_INVALID_STATE if esp_vfs_fat_rawflash_mount was already called for the same partition
- ESP_ERR_NO_MEM if memory can not be allocated
- ESP_FAIL if partition can not be mounted
- other error codes from SPI flash driver, or FATFS drivers

Parameters

- base_path: path where FATFS partition should be mounted (e.g. “/spiflash”)
- partition_label: label of the partition which should be used
- mount_config: pointer to structure with extra parameters for mounting FATFS

```
esp_err_t esp_vfs_fat_rawflash_unmount (const char *base_path, const char
                                       *partition_label)
```

Unmount FAT filesystem and release resources acquired using esp_vfs_fat_rawflash_mount.

Return

- ESP_OK on success
- ESP_ERR_INVALID_STATE if esp_vfs_fat_spiflash_mount hasn't been called

Parameters

- base_path: path where partition should be registered (e.g. “/spiflash”)
- partition_label: label of partition to be unmounted

FatFS disk IO layer

FatFs has been extended with API functions that register the disk I/O driver at runtime.

They provide implementation of disk I/O functions for SD/MMC cards and can be registered for the given FatFs drive number using the function `ff_diskio_register_sdmmc()`.

```
void ff_diskio_register (BYTE pdrv, const ff_diskio_impl_t *discio_impl)
```

Register or unregister diskio driver for given drive number.

When FATFS library calls one of disk_xxx functions for driver number pdrv, corresponding function in discio_impl for given pdrv will be called.

Parameters

- pdrv: drive number
- discio_impl: pointer to `ff_diskio_impl_t` structure with diskio functions or NULL to unregister and free previously registered drive

```
struct ff_diskio_impl_t
```

Structure of pointers to disk IO driver functions.

See FatFs documentation for details about these functions

Public Members

DSTATUS (***init**) (unsigned char pdrv)
disk initialization function

DSTATUS (***status**) (unsigned char pdrv)
disk status check function

DRESULT (***read**) (unsigned char pdrv, unsigned char *buff, uint32_t sector, unsigned count)
sector read function

DRESULT (***write**) (unsigned char pdrv, **const** unsigned char *buff, uint32_t sector, unsigned count)
sector write function

DRESULT (***ioctl**) (unsigned char pdrv, unsigned char cmd, void *buff)
function to get info about disk and do some misc operations

void **ff_diskio_register_sdmmc** (unsigned char pdrv, *sdmmc_card_t* *card)
Register SD/MMC diskio driver

Parameters

- pdrv: drive number
- card: pointer to *sdmmc_card_t* structure describing a card; card should be initialized before calling `f_mount`.

esp_err_t **ff_diskio_register_wl_partition** (unsigned char pdrv, *wl_handle_t* flash_handle)
Register spi flash partition

Parameters

- pdrv: drive number
- flash_handle: handle of the wear levelling partition.

esp_err_t **ff_diskio_register_raw_partition** (unsigned char pdrv, **const** *esp_partition_t* *part_handle)
Register spi flash partition

Parameters

- pdrv: drive number
- part_handle: pointer to raw flash partition.

2.6.2 Manufacturing Utility

Introduction

This utility is designed to create instances of factory NVS partition images on a per-device basis for mass manufacturing purposes. The NVS partition images are created from CSV files containing user-provided configurations and values.

Please note that this utility only creates manufacturing binary images which then need to be flashed onto your devices using:

- esptool.py
- Flash Download tool (available on Windows only)
- Direct flash programming

Prerequisites

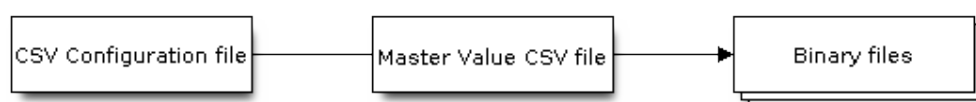
This utility is dependent on `esp-idf`'s NVS partition utility.

- **Operating System requirements:**
 - Linux / MacOS / Windows (standard distributions)
- **The following packages are needed to use this utility:**

– Python: <<https://www.python.org/downloads/>>

Note:**Before using this utility, please make sure that:**

- The path to Python is added to the PATH environment variable.
 - You have installed the packages from *requirement.txt*, the file in the root of the esp-idf directory.
-

Workflow**CSV Configuration File**

This file contains the configuration of the device to be flashed.

The data in the configuration file has the following format (the *REPEAT* tag is optional):

```
name1,namespace,    <-- First entry should be of type "namespace"
key1,type1,encoding1
key2,type2,encoding2,REPEAT
name2,namespace,
key3,type3,encoding3
key4,type4,encoding4
```

Note: The first line in this file should always be the namespace entry.

Each line should have three parameters: *key*, *type*, *encoding*, separated by a comma. If the *REPEAT* tag is present, the value corresponding to this key in the master value CSV file will be the same for all devices.

Please refer to README of the NVS Partition Generator utility for detailed description of each parameter.

Below is a sample example of such a configuration file:

```
app,namespace,
firmware_key,data,hex2bin
serial_no,data,string,REPEAT
device_no,data,i32
```

Note:**Make sure there are no spaces:**

- before and after ‘,’
 - at the end of each line in a CSV file
-

Master Value CSV File

This file contains details of the devices to be flashed. Each line in this file corresponds to a device instance.

The data in the master value CSV file has the following format:

```
key1, key2, key3, .....
value1, value2, value3, ....
```

Note: The first line in the file should always contain the `key` names. All the keys from the configuration file should be present here in the **same order**. This file can have additional columns (keys). The additional keys will be treated as metadata and would not be part of the final binary files.

Each line should contain the `value` of the corresponding keys, separated by a comma. If the key has the `REPEAT` tag, its corresponding value **must** be entered in the second line only. Keep the entry empty for this value in the following lines.

The description of this parameter is as follows:

value Data value

Data value is the value of data corresponding to the key.

Below is a sample example of a master value CSV file:

```
id, firmware_key, serial_no, device_no
1, 1a2b3c4d5e6faabb, A1, 101
2, 1a2b3c4d5e6fccdd, , 102
3, 1a2b3c4d5e6feeff, , 103
```

Note: If the `'REPEAT'` tag is present, a new master value CSV file will be created in the same folder as the input Master CSV File with the values inserted at each line for the key with the `'REPEAT'` tag.

This utility creates intermediate CSV files which are used as input for the NVS partition utility to generate the binary files.

The format of this intermediate CSV file is as follows:

```
key, type, encoding, value
key, namespace, ,
key1, type1, encoding1, value1
key2, type2, encoding2, value2
```

An instance of an intermediate CSV file will be created for each device on an individual basis.

Running the utility

Usage:

```
python mfg_gen.py [-h] {generate, generate-key} ...
```

Optional Arguments:

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
↪-----+
| No. | Parameter | Description |
↪      |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1   | -h, --help | show this help message and exit |
↪      |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
↪-----+

```

Commands:

(continues on next page)

(continued from previous page)

```

Run mfg_gen.py {command} -h for additional help
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
↪-----+
| No. | Parameter | Description |
↪-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | generate | Generate NVS partition |
↪-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
↪-----+
| 2 | generate-key | Generate keys for encryption |
↪-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
↪-----+

```

To generate factory images for each device (Default): Usage:

```

python mfg_gen.py generate [-h] [--fileid FILEID] [--version {1,2}] [--keygen]
                          [--keyfile KEYFILE] [--inputkey INPUTKEY]
                          [--outdir OUTDIR]
                          conf values prefix size

Positional Arguments:
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
↪-----+
| Parameter | Description |
↪-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| conf | Path to configuration csv file to parse |
↪-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| values | Path to values csv file to parse |
↪-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| prefix | Unique name for each output filename prefix |
↪-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| size | Size of NVS partition in bytes |
↪-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| | (must be multiple of 4096) |
↪-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
↪-----+

Optional Arguments:
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
↪-----+
| Parameter | Description |
↪-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| -h, --help | show this help message and exit |
↪-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| --fileid FILEID | Unique file identifier(any key in values file) |
↪-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| | for each filename suffix (Default: numeric value(1,
↪2,3...)) |

```

(continues on next page)

(continued from previous page)

| | |
|----------------------------------|---|
| <code>--version {1,2}</code> | Set multipage blob version. |
| | Version 1 - Multipage blob support disabled. |
| | Version 2 - Multipage blob support enabled. |
| | Default: Version 2 |
| <code>--keygen</code> | Generates key for encrypting NVS partition |
| <code>--inputkey INPUTKEY</code> | File having key for encrypting NVS partition |
| <code>--outdir OUTDIR</code> | Output directory to store files created (Default: current directory) |

You can run the utility to generate factory images for each device using the command below. A sample CSV file is provided with the utility:

```
python mfg_gen.py generate samples/sample_config.csmples/sample_values_singlepage_
↳blob.csv Sample 0x3000
```

The master value CSV file should have the path in the `file` type relative to the directory from which you are running the utility.

To generate encrypted factory images for each device:

You can run the utility to encrypt factory images for each device using the command below. A sample CSV file is provided with the utility:

- Encrypt by allowing the utility to generate encryption keys:

```
python mfg_gen.py generate samples/sample_config.csv samples/sample_values_
↳singlepage_blob.csv Sample 0x3000 --keygen
```

Note: Encryption key of the following format `<outdir>/keys/keys-<prefix>-<fileid>.bin` is created.

Note: This newly created file having encryption keys in `keys/` directory is compatible with NVS key-partition structure. Refer to [NVS key partition](#) for more details.

- Encrypt by providing the encryption keys as input binary file:

```
python mfg_gen.py generate samples/sample_config.csv samples/sample_values_
↳singlepage_blob.csv Sample 0x3000 --inputkey keys/sample_keys.bin
```

To generate only encryption keys: Usage:

```
python mfg_gen.py generate-key [-h] [--keyfile KEYFILE] [--outdir OUTDIR]
```

Optional Arguments:

| Parameter | Description |
|--------------------------------|--|
| <code>-h, --help</code> | show this help message and exit |
| <code>--keyfile KEYFILE</code> | Path to output encryption keys file |
| <code>--outdir OUTDIR</code> | Output directory to store files created. (Default: current directory) |

You can run the utility to generate only encryption keys using the command below:

```
python mfg_gen.py generate-key
```

Note: Encryption key of the following format `<outdir>/keys/keys-<timestamp>.bin` is created. Timestamp format is: `%m-%d_%H-%M`.

Note: To provide custom target filename use the `-keyfile` argument.

Generated encryption key binary file can further be used to encrypt factory images created on the per device basis.

The default numeric value: 1,2,3... of the `fileid` argument corresponds to each line bearing device instance values in the master value CSV file.

While running the manufacturing utility, the following folders will be created in the specified `outdir` directory:

- `bin/` for storing the generated binary files
- `csv/` for storing the generated intermediate CSV files
- `keys/` for storing encryption keys (when generating encrypted factory images)

2.6.3 Non-volatile storage library

Introduction

Non-volatile storage (NVS) library is designed to store key-value pairs in flash. This section introduces some concepts used by NVS.

Underlying storage Currently, NVS uses a portion of main flash memory through `spi_flash_{read|write|erase}` APIs. The library uses all the partitions with `data` type and `nvs` subtype. The application can choose to use the partition with the label `nvs` through the `nvs_open` API function or any other partition by specifying its name using the `nvs_open_from_part` API function.

Future versions of this library may have other storage backends to keep data in another flash chip (SPI or I2C), RTC, FRAM, etc.

Note: if an NVS partition is truncated (for example, when the partition table layout is changed), its contents should be erased. ESP-IDF build system provides a `idf.py erase_flash` target to erase all contents of the flash chip.

Note: NVS works best for storing many small values, rather than a few large values of the type 'string' and 'blob'. If you need to store large blobs or strings, consider using the facilities provided by the FAT filesystem on top of the wear levelling library.

Keys and values NVS operates on key-value pairs. Keys are ASCII strings; the maximum key length is currently 15 characters. Values can have one of the following types:

- integer types: `uint8_t`, `int8_t`, `uint16_t`, `int16_t`, `uint32_t`, `int32_t`, `uint64_t`, `int64_t`
- zero-terminated string
- variable length binary data (blob)

Note: String values are currently limited to 4000 bytes. This includes the null terminator. Blob values are limited to 508000 bytes or 97.6% of the partition size - 4000 bytes, whichever is lower.

Additional types, such as `float` and `double` might be added later.

Keys are required to be unique. Assigning a new value to an existing key works as follows:

- if the new value is of the same type as the old one, value is updated
- if the new value has a different data type, an error is returned

Data type check is also performed when reading a value. An error is returned if the data type of the read operation does not match the data type of the value.

Namespaces To mitigate potential conflicts in key names between different components, NVS assigns each key-value pair to one of namespaces. Namespace names follow the same rules as key names, i.e., the maximum length is 15 characters. Namespace name is specified in the `nvs_open` or `nvs_open_from_part` call. This call returns an opaque handle, which is used in subsequent calls to the `nvs_get_*`, `nvs_set_*`, and `nvs_commit` functions. This way, a handle is associated with a namespace, and key names will not collide with same names in other namespaces. Please note that the namespaces with the same name in different NVS partitions are considered as separate namespaces.

Security, tampering, and robustness NVS is not directly compatible with the `{IDF_TARGET_NAME}` flash encryption system. However, data can still be stored in encrypted form if NVS encryption is used together with `{IDF_TARGET_NAME}` flash encryption. Please refer to [NVS Encryption](#) for more details.

If NVS encryption is not used, it is possible for anyone with physical access to the flash chip to alter, erase, or add key-value pairs. With NVS encryption enabled, it is not possible to alter or add a key-value pair and get recognized as a valid pair without knowing corresponding NVS encryption keys. However, there is no tamper-resistance against the erase operation.

The library does try to recover from conditions when flash memory is in an inconsistent state. In particular, one should be able to power off the device at any point and time and then power it back on. This should not result in loss of data, except for the new key-value pair if it was being written at the moment of powering off. The library should also be able to initialize properly with any random data present in flash memory.

Internals

Log of key-value pairs NVS stores key-value pairs sequentially, with new key-value pairs being added at the end. When a value of any given key has to be updated, a new key-value pair is added at the end of the log and the old key-value pair is marked as erased.

Pages and entries NVS library uses two main entities in its operation: pages and entries. Page is a logical structure which stores a portion of the overall log. Logical page corresponds to one physical sector of flash memory. Pages which are in use have a *sequence number* associated with them. Sequence numbers impose an ordering on pages. Higher sequence numbers correspond to pages which were created later. Each page can be in one of the following states:

Empty/uninitialized Flash storage for the page is empty (all bytes are `0xff`). Page is not used to store any data at this point and does not have a sequence number.

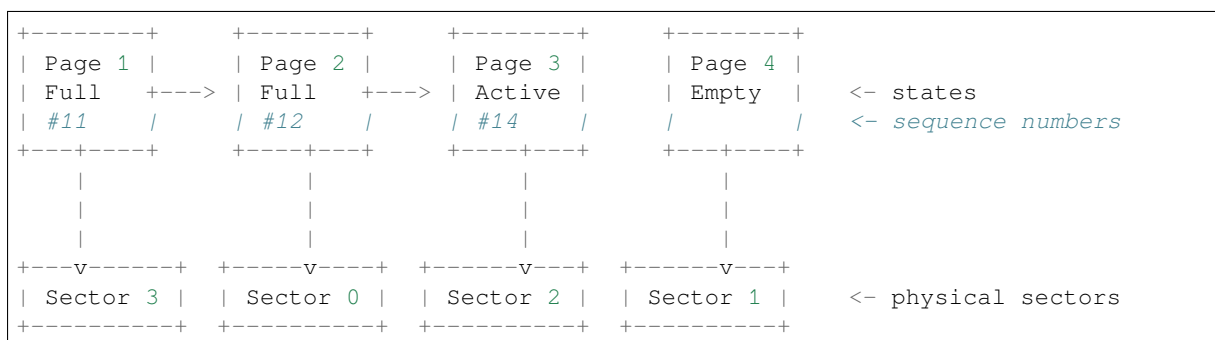
Active Flash storage is initialized, page header has been written to flash, page has a valid sequence number. Page has some empty entries and data can be written there. No more than one page can be in this state at any given moment.

Full Flash storage is in a consistent state and is filled with key-value pairs. Writing new key-value pairs into this page is not possible. It is still possible to mark some key-value pairs as erased.

Erasing Non-erased key-value pairs are being moved into another page so that the current page can be erased. This is a transient state, i.e., page should never stay in this state at the time when any API call returns. In case of a sudden power off, the move-and-erase process will be completed upon the next power-on.

Corrupted Page header contains invalid data, and further parsing of page data was canceled. Any items previously written into this page will not be accessible. The corresponding flash sector will not be erased immediately and will be kept along with sectors in *uninitialized* state for later use. This may be useful for debugging.

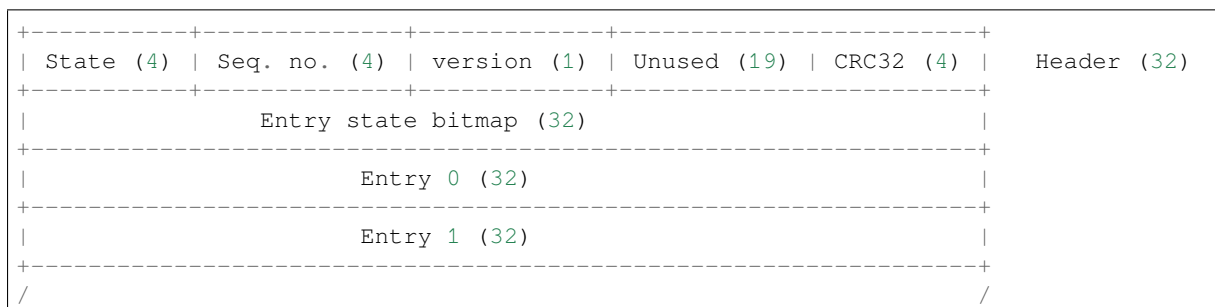
Mapping from flash sectors to logical pages does not have any particular order. The library will inspect sequence numbers of pages found in each flash sector and organize pages in a list based on these numbers.



Structure of a page For now, we assume that flash sector size is 4096 bytes and that `{IDF_TARGET_NAME}` flash encryption hardware operates on 32-byte blocks. It is possible to introduce some settings configurable at compile-time (e.g., via `menuconfig`) to accommodate flash chips with different sector sizes (although it is not clear if other components in the system, e.g., SPI flash driver and SPI flash cache can support these other sizes).

Page consists of three parts: header, entry state bitmap, and entries themselves. To be compatible with `{IDF_TARGET_NAME}` flash encryption, entry size is 32 bytes. For integer types, entry holds one key-value pair. For strings and blobs, an entry holds part of key-value pair (more on that in the entry structure description).

The following diagram illustrates the page structure. Numbers in parentheses indicate the size of each part in bytes.



(continues on next page)

(continued from previous page)



Page header and entry state bitmap are always written to flash unencrypted. Entries are encrypted if flash encryption feature of {IDF_TARGET_NAME} is used.

Page state values are defined in such a way that changing state is possible by writing 0 into some of the bits. Therefore it is not necessary to erase the page to change its state unless that is a change to the *erased* state.

The version field in the header reflects the NVS format version used. For backward compatibility reasons, it is decremented for every version upgrade starting at 0xff (i.e., 0xff for version-1, 0xfe for version-2 and so on).

CRC32 value in the header is calculated over the part which does not include a state value (bytes 4 to 28). The unused part is currently filled with 0xff bytes.

The following sections describe the structure of entry state bitmap and entry itself.

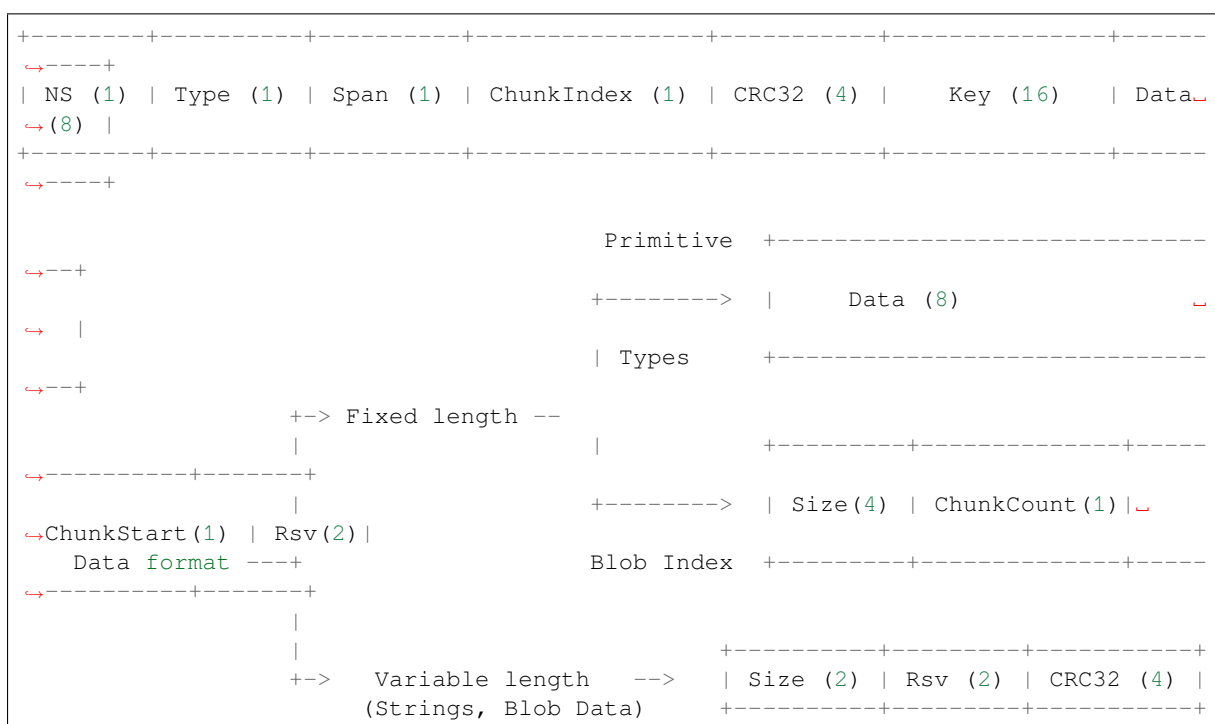
Entry and entry state bitmap Each entry can be in one of the following three states represented with two bits in the entry state bitmap. The final four bits in the bitmap (256 - 2 * 126) are not used.

Empty (2' b11) Nothing is written into the specific entry yet. It is in an uninitialized state (all bytes are 0xff).

Written (2' b10) A key-value pair (or part of key-value pair which spans multiple entries) has been written into the entry.

Erased (2' b00) A key-value pair in this entry has been discarded. Contents of this entry will not be parsed anymore.

Structure of entry For values of primitive types (currently integers from 1 to 8 bytes long), entry holds one key-value pair. For string and blob types, entry holds part of the whole key-value pair. For strings, in case when a key-value pair spans multiple entries, all entries are stored in the same page. Blobs are allowed to span over multiple pages by dividing them into smaller chunks. For tracking these chunks, an additional fixed length metadata entry is stored called “blob index”. Earlier formats of blobs are still supported (can be read and modified). However, once the blobs are modified, they are stored using the new format.



Individual fields in entry structure have the following meanings:

NS Namespace index for this entry. For more information on this value, see the section on namespaces implementation.

Type One byte indicating the value data type. See the `ItemType` enumeration in `nvs_flash/include/nvs_handle.hpp` for possible values.

Span Number of entries used by this key-value pair. For integer types, this is equal to 1. For strings and blobs, this depends on value length.

ChunkIndex Used to store the index of a blob-data chunk for blob types. For other types, this should be `0xff`.

CRC32 Checksum calculated over all the bytes in this entry, except for the CRC32 field itself.

Key Zero-terminated ASCII string containing a key name. Maximum string length is 15 bytes, excluding a zero terminator.

Data For integer types, this field contains the value itself. If the value itself is shorter than 8 bytes, it is padded to the right, with unused bytes filled with `0xff`.

For “blob index” entry, these 8 bytes hold the following information about data-chunks:

- **Size** (Only for blob index.) Size, in bytes, of complete blob data.
- **ChunkCount** (Only for blob index.) Total number of blob-data chunks into which the blob was divided during storage.
- **ChunkStart** (Only for blob index.) `ChunkIndex` of the first blob-data chunk of this blob. Subsequent chunks have `chunkIndex` incrementally allocated (step of 1).

For string and blob data chunks, these 8 bytes hold additional data about the value, which are described below:

- **Size** (Only for strings and blobs.) Size, in bytes, of actual data. For strings, this includes zero terminators.
- **CRC32** (Only for strings and blobs.) Checksum calculated over all bytes of data.

Variable length values (strings and blobs) are written into subsequent entries, 32 bytes per entry. The *Span* field of the first entry indicates how many entries are used.

Namespaces As mentioned above, each key-value pair belongs to one of the namespaces. Namespace identifiers (strings) are stored as keys of key-value pairs in namespace with index 0. Values corresponding to these keys are indexes of these namespaces.

```
+-----+
| NS=0 Type=uint8_t Key="wifi" Value=1 | Entry describing namespace "wifi"
+-----+
| NS=1 Type=uint32_t Key="channel" Value=6 | Key "channel" in namespace "wifi"
+-----+
| NS=0 Type=uint8_t Key="pwm" Value=2 | Entry describing namespace "pwm"
+-----+
| NS=2 Type=uint16_t Key="channel" Value=20 | Key "channel" in namespace "pwm"
+-----+
```

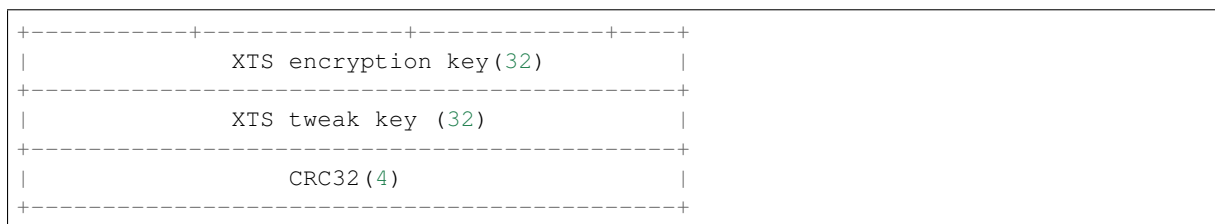
Item hash list To reduce the number of reads from flash memory, each member of the `Page` class maintains a list of pairs: item index; item hash. This list makes searches much quicker. Instead of iterating over all entries, reading them from flash one at a time, `Page::findItem` first performs a search for the item hash in the hash list. This gives the item index within the page if such an item exists. Due to a hash collision, it is possible that a different item will be found. This is handled by falling back to iteration over items in flash.

Each node in the hash list contains a 24-bit hash and 8-bit item index. Hash is calculated based on item namespace, key name, and `ChunkIndex`. CRC32 is used for calculation; the result is truncated to 24 bits. To reduce the overhead for storing 32-bit entries in a linked list, the list is implemented as a double-linked list of arrays. Each array holds 29 entries, for the total size of 128 bytes, together with linked list pointers and a 32-bit count field. The minimum amount of extra RAM usage per page is therefore 128 bytes; maximum is 640 bytes.

NVS Encryption

Data stored in NVS partitions can be encrypted using AES-XTS in the manner similar to the one mentioned in disk encryption standard IEEE P1619. For the purpose of encryption, each entry is treated as one *sector* and relative address of the entry (w.r.t. partition-start) is fed to the encryption algorithm as *sector-number*. The keys required for NVS encryption are stored in yet another partition, which is protected using *Flash Encryption*. Therefore, enabling *Flash Encryption* is a prerequisite for NVS encryption.

NVS key partition An application requiring NVS encryption support needs to be compiled with a key-partition of the type *data* and subtype *key*. This partition should be marked as *encrypted*. Refer to *Partition Tables* for more details. The size of the partition should be 4096 bytes (minimum partition size). The structure of this partition is depicted below.



This partition can be generated using *nvs partition generator* utility and flashed onto the device. Since the partition is marked *encrypted* and *Flash Encryption* is enabled, bootloader will encrypt this partition using flash encryption key on the first boot. Alternatively, the keys can be generated after startup using the `nvs_flash_generate_keys` API function provided by `nvs_flash.h`, which will then write those keys onto the key-partition in encrypted form.

It is possible for an application to use different keys for different NVS partitions and thereby have multiple key-partitions. However, it is a responsibility of the application to provide correct key-partition/keys for the purpose of encryption/decryption.

Encrypted Read/Write The same NVS API functions `nvs_get_*` or `nvs_set_*` can be used for reading of, and writing to an encrypted nvs partition as well. However, the API functions for initialising NVS partitions are different: `nvs_flash_secure_init` and `nvs_flash_secure_init_partition` instead of `nvs_flash_init` and `nvs_flash_init_partition` respectively. The `nvs_sec_cfg_t` structure required for these API functions can be populated using `nvs_flash_generate_keys` or `nvs_flash_read_security_cfg`.

Applications are expected to follow the steps below in order to perform NVS read/write operations with encryption enabled.

1. Find key partition and NVS data partition using `esp_partition_find*` API functions.
2. Populate the `nvs_sec_cfg_t` struct using the `nvs_flash_read_security_cfg` or `nvs_flash_generate_keys` API functions.
3. Initialise NVS flash partition using the `nvs_flash_secure_init` or `nvs_flash_secure_init_partition` API functions.
4. Open a namespace using the `nvs_open` or `nvs_open_from_part` API functions.
5. Perform NVS read/write operations using `nvs_get_*` or `nvs_set_*`.
6. Deinitialise an NVS partition using `nvs_flash_deinit`.

NVS iterators Iterators allow to list key-value pairs stored in NVS, based on specified partition name, namespace, and data type.

There are the following functions available:

- `nvs_entry_find` returns an opaque handle, which is used in subsequent calls to the `nvs_entry_next` and `nvs_entry_info` functions.
- `nvs_entry_next` returns iterator to the next key-value pair.
- `nvs_entry_info` returns information about each key-value pair

If none or no other key-value pair was found for given criteria, `nvs_entry_find` and `nvs_entry_next` return NULL. In that case, the iterator does not have to be released. If the iterator is no longer needed, you can release it by using the function `nvs_release_iterator`.

NVS Partition Generator Utility

This utility helps generate NVS partition binary files which can be flashed separately on a dedicated partition via a flashing utility. Key-value pairs to be flashed onto the partition can be provided via a CSV file. For more details, please refer to *NVS Partition Generator Utility*.

Application Example

You can find two code examples in the [storage](#) directory of ESP-IDF examples:

[storage/nvs_rw_value](#)

Demonstrates how to read a single integer value from, and write it to NVS.

The value checked in this example holds the number of the ESP32 module restarts. The value's function as a counter is only possible due to its storing in NVS.

The example also shows how to check if a read / write operation was successful, or if a certain value has not been initialized in NVS. The diagnostic procedure is provided in plain text to help you track the program flow and capture any issues on the way.

[storage/nvs_rw_blob](#)

Demonstrates how to read a single integer value and a blob (binary large object), and write them to NVS to preserve this value between ESP32 module restarts.

- value - tracks the number of the ESP32 module soft and hard restarts.
- blob - contains a table with module run times. The table is read from NVS to dynamically allocated RAM. A new run time is added to the table on each manually triggered soft restart, and then the added run time is written to NVS. Triggering is done by pulling down GPIO0.

The example also shows how to implement the diagnostic procedure to check if the read / write operation was successful.

[storage/nvs_rw_value_cxx](#)

This example does exactly the same as [storage/nvs_rw_value](#), except that it uses the C++ nvs handle class.

API Reference

Header File

- [nvs_flash/include/nvs_flash.h](#)

Functions

esp_err_t **nvs_flash_init** (void)

Initialize the default NVS partition.

This API initialises the default NVS partition. The default NVS partition is the one that is labeled “nvs” in the partition table.

Return

- ESP_OK if storage was successfully initialized.
- ESP_ERR_NVS_NO_FREE_PAGES if the NVS storage contains no empty pages (which may happen if NVS partition was truncated)
- ESP_ERR_NOT_FOUND if no partition with label “nvs” is found in the partition table
- one of the error codes from the underlying flash storage driver

esp_err_t **nvs_flash_init_partition** (const char **partition_label*)

Initialize NVS flash storage for the specified partition.

Return

- ESP_OK if storage was successfully initialized.
- ESP_ERR_NVS_NO_FREE_PAGES if the NVS storage contains no empty pages (which may happen if NVS partition was truncated)
- ESP_ERR_NOT_FOUND if specified partition is not found in the partition table
- one of the error codes from the underlying flash storage driver

Parameters

- [in] *partition_label*: Label of the partition. Must be no longer than 16 characters.

esp_err_t **nvs_flash_init_partition_ptr** (const *esp_partition_t* *partition)

Initialize NVS flash storage for the partition specified by partition pointer.

Return

- ESP_OK if storage was successfully initialized
- ESP_ERR_NVS_NO_FREE_PAGES if the NVS storage contains no empty pages (which may happen if NVS partition was truncated)
- ESP_ERR_INVALID_ARG in case partition is NULL
- one of the error codes from the underlying flash storage driver

Parameters

- [in] partition: pointer to a partition obtained by the ESP partition API.

esp_err_t **nvs_flash_deinit** (void)

Deinitialize NVS storage for the default NVS partition.

Default NVS partition is the partition with “nvs” label in the partition table.

Return

- ESP_OK on success (storage was deinitialized)
- ESP_ERR_NVS_NOT_INITIALIZED if the storage was not initialized prior to this call

esp_err_t **nvs_flash_deinit_partition** (const char *partition_label)

Deinitialize NVS storage for the given NVS partition.

Return

- ESP_OK on success
- ESP_ERR_NVS_NOT_INITIALIZED if the storage for given partition was not initialized prior to this call

Parameters

- [in] partition_label: Label of the partition

esp_err_t **nvs_flash_erase** (void)

Erase the default NVS partition.

Erases all contents of the default NVS partition (one with label “nvs”).

Note If the partition is initialized, this function first de-initializes it. Afterwards, the partition has to be initialized again to be used.

Return

- ESP_OK on success
- ESP_ERR_NOT_FOUND if there is no NVS partition labeled “nvs” in the partition table
- different error in case de-initialization fails (shouldn’ t happen)

esp_err_t **nvs_flash_erase_partition** (const char *part_name)

Erase specified NVS partition.

Erase all content of a specified NVS partition

Note If the partition is initialized, this function first de-initializes it. Afterwards, the partition has to be initialized again to be used.

Return

- ESP_OK on success
- ESP_ERR_NOT_FOUND if there is no NVS partition with the specified name in the partition table
- different error in case de-initialization fails (shouldn’ t happen)

Parameters

- [in] part_name: Name (label) of the partition which should be erased

esp_err_t **nvs_flash_erase_partition_ptr** (const *esp_partition_t* *partition)

Erase custom partition.

Erase all content of specified custom partition.

Note If the partition is initialized, this function first de-initializes it. Afterwards, the partition has to be initialized again to be used.

Return

- ESP_OK on success

- ESP_ERR_NOT_FOUND if there is no partition with the specified parameters in the partition table
- ESP_ERR_INVALID_ARG in case partition is NULL
- one of the error codes from the underlying flash storage driver

Parameters

- [in] *partition*: pointer to a partition obtained by the ESP partition API.

esp_err_t **nvs_flash_secure_init** (*nvs_sec_cfg_t* **cfg*)

Initialize the default NVS partition.

This API initialises the default NVS partition. The default NVS partition is the one that is labeled “nvs” in the partition table.

Return

- ESP_OK if storage was successfully initialized.
- ESP_ERR_NVS_NO_FREE_PAGES if the NVS storage contains no empty pages (which may happen if NVS partition was truncated)
- ESP_ERR_NOT_FOUND if no partition with label “nvs” is found in the partition table
- one of the error codes from the underlying flash storage driver

Parameters

- [in] *cfg*: Security configuration (keys) to be used for NVS encryption/decryption. If *cfg* is NULL, no encryption is used.

esp_err_t **nvs_flash_secure_init_partition** (**const** char **partition_label*, *nvs_sec_cfg_t* **cfg*)

Initialize NVS flash storage for the specified partition.

Return

- ESP_OK if storage was successfully initialized.
- ESP_ERR_NVS_NO_FREE_PAGES if the NVS storage contains no empty pages (which may happen if NVS partition was truncated)
- ESP_ERR_NOT_FOUND if specified partition is not found in the partition table
- one of the error codes from the underlying flash storage driver

Parameters

- [in] *partition_label*: Label of the partition. Note that internally a reference to passed value is kept and it should be accessible for future operations
- [in] *cfg*: Security configuration (keys) to be used for NVS encryption/decryption. If *cfg* is null, no encryption/decryption is used.

esp_err_t **nvs_flash_generate_keys** (**const** *esp_partition_t* **partition*, *nvs_sec_cfg_t* **cfg*)

Generate and store NVS keys in the provided esp partition.

Return -ESP_OK, if *cfg* was read successfully; -or error codes from *esp_partition_write/erase* APIs.

Parameters

- [in] *partition*: Pointer to partition structure obtained using *esp_partition_find_first* or *esp_partition_get*. Must be non-NULL.
- [out] *cfg*: Pointer to nvs security configuration structure. Pointer must be non-NULL. Generated keys will be populated in this structure.

esp_err_t **nvs_flash_read_security_cfg** (**const** *esp_partition_t* **partition*, *nvs_sec_cfg_t* **cfg*)

Read NVS security configuration from a partition.

Note Provided partition is assumed to be marked ‘encrypted’ .

Return -ESP_OK, if *cfg* was read successfully; -ESP_ERR_NVS_KEYS_NOT_INITIALIZED, if the partition is not yet written with keys. -ESP_ERR_NVS_CORRUPT_KEY_PART, if the partition containing keys is found to be corrupt -or error codes from *esp_partition_read* API.

Parameters

- [in] *partition*: Pointer to partition structure obtained using *esp_partition_find_first* or *esp_partition_get*. Must be non-NULL.
- [out] *cfg*: Pointer to nvs security configuration structure. Pointer must be non-NULL.

Structures

struct *nvs_sec_cfg_t*

Key for encryption and decryption.

Public Members

`uint8_t eky[NVS_KEY_SIZE]`
XTS encryption and decryption key

`uint8_t tky[NVS_KEY_SIZE]`
XTS tweak key

Macros

`NVS_KEY_SIZE`

Header File

- [nvs_flash/include/nvs.h](#)

Functions

`esp_err_t nvs_set_i8(nvs_handle_t handle, const char *key, int8_t value)`
set int8_t value for given key

Set value for the key, given its name. Note that the actual storage will not be updated until `nvs_commit` is called.

Return

- `ESP_OK` if value was set successfully
- `ESP_ERR_NVS_INVALID_HANDLE` if handle has been closed or is `NULL`
- `ESP_ERR_NVS_READ_ONLY` if storage handle was opened as read only
- `ESP_ERR_NVS_INVALID_NAME` if key name doesn't satisfy constraints
- `ESP_ERR_NVS_NOT_ENOUGH_SPACE` if there is not enough space in the underlying storage to save the value
- `ESP_ERR_NVS_REMOVE_FAILED` if the value wasn't updated because flash write operation has failed. The value was written however, and update will be finished after re-initialization of `nvs`, provided that flash operation doesn't fail again.

Parameters

- [in] `handle`: Handle obtained from `nvs_open` function. Handles that were opened read only cannot be used.
- [in] `key`: Key name. Maximal length is determined by the underlying implementation, but is guaranteed to be at least 15 characters. Shouldn't be empty.
- [in] `value`: The value to set.

`esp_err_t nvs_set_u8(nvs_handle_t handle, const char *key, uint8_t value)`
set uint8_t value for given key

This function is the same as `nvs_set_i8` except for the data type.

`esp_err_t nvs_set_i16(nvs_handle_t handle, const char *key, int16_t value)`
set int16_t value for given key

This function is the same as `nvs_set_i8` except for the data type.

`esp_err_t nvs_set_u16(nvs_handle_t handle, const char *key, uint16_t value)`
set uint16_t value for given key

This function is the same as `nvs_set_i8` except for the data type.

`esp_err_t nvs_set_i32(nvs_handle_t handle, const char *key, int32_t value)`
set int32_t value for given key

This function is the same as `nvs_set_i8` except for the data type.

`esp_err_t nvs_set_u32(nvs_handle_t handle, const char *key, uint32_t value)`
set uint32_t value for given key

This function is the same as `nvs_set_i8` except for the data type.

esp_err_t **nvs_set_i64** (*nvs_handle_t* handle, const char *key, int64_t value)
set int64_t value for given key

This function is the same as `nvs_set_i8` except for the data type.

esp_err_t **nvs_set_u64** (*nvs_handle_t* handle, const char *key, uint64_t value)
set uint64_t value for given key

This function is the same as `nvs_set_i8` except for the data type.

esp_err_t **nvs_set_str** (*nvs_handle_t* handle, const char *key, const char *value)
set string for given key

Set value for the key, given its name. Note that the actual storage will not be updated until `nvs_commit` is called.

Return

- `ESP_OK` if value was set successfully
- `ESP_ERR_NVS_INVALID_HANDLE` if handle has been closed or is `NULL`
- `ESP_ERR_NVS_READ_ONLY` if storage handle was opened as read only
- `ESP_ERR_NVS_INVALID_NAME` if key name doesn't satisfy constraints
- `ESP_ERR_NVS_NOT_ENOUGH_SPACE` if there is not enough space in the underlying storage to save the value
- `ESP_ERR_NVS_REMOVE_FAILED` if the value wasn't updated because flash write operation has failed. The value was written however, and update will be finished after re-initialization of `nvs`, provided that flash operation doesn't fail again.
- `ESP_ERR_NVS_VALUE_TOO_LONG` if the string value is too long

Parameters

- [in] handle: Handle obtained from `nvs_open` function. Handles that were opened read only cannot be used.
- [in] key: Key name. Maximal length is `(NVS_KEY_NAME_MAX_SIZE-1)` characters. Shouldn't be empty.
- [in] value: The value to set. For strings, the maximum length (including null character) is 4000 bytes, if there is one complete page free for writing. This decreases, however, if the free space is fragmented.

esp_err_t **nvs_get_i8** (*nvs_handle_t* handle, const char *key, int8_t *out_value)
get int8_t value for given key

These functions retrieve value for the key, given its name. If `key` does not exist, or the requested variable type doesn't match the type which was used when setting a value, an error is returned.

In case of any error, `out_value` is not modified.

`out_value` has to be a pointer to an already allocated variable of the given type.

```
// Example of using nvs_get_i32:
int32_t max_buffer_size = 4096; // default value
esp_err_t err = nvs_get_i32(my_handle, "max_buffer_size", &max_buffer_size);
assert(err == ESP_OK || err == ESP_ERR_NVS_NOT_FOUND);
// if ESP_ERR_NVS_NOT_FOUND was returned, max_buffer_size will still
// have its default value.
```

Return

- `ESP_OK` if the value was retrieved successfully
- `ESP_ERR_NVS_NOT_FOUND` if the requested key doesn't exist
- `ESP_ERR_NVS_INVALID_HANDLE` if handle has been closed or is `NULL`
- `ESP_ERR_NVS_INVALID_NAME` if key name doesn't satisfy constraints
- `ESP_ERR_NVS_INVALID_LENGTH` if length is not sufficient to store data

Parameters

- [in] handle: Handle obtained from `nvs_open` function.
- [in] key: Key name. Maximal length is determined by the underlying implementation, but is guaranteed to be at least 15 characters. Shouldn't be empty.

- `out_value`: Pointer to the output value. May be NULL for `nvs_get_str` and `nvs_get_blob`, in this case required length will be returned in length argument.

`esp_err_t nvs_get_u8` (*nvs_handle_t* handle, **const** char *key, uint8_t *out_value)
get uint8_t value for given key

This function is the same as `nvs_get_i8` except for the data type.

`esp_err_t nvs_get_i16` (*nvs_handle_t* handle, **const** char *key, int16_t *out_value)
get int16_t value for given key

This function is the same as `nvs_get_i8` except for the data type.

`esp_err_t nvs_get_u16` (*nvs_handle_t* handle, **const** char *key, uint16_t *out_value)
get uint16_t value for given key

This function is the same as `nvs_get_i8` except for the data type.

`esp_err_t nvs_get_i32` (*nvs_handle_t* handle, **const** char *key, int32_t *out_value)
get int32_t value for given key

This function is the same as `nvs_get_i8` except for the data type.

`esp_err_t nvs_get_u32` (*nvs_handle_t* handle, **const** char *key, uint32_t *out_value)
get uint32_t value for given key

This function is the same as `nvs_get_i8` except for the data type.

`esp_err_t nvs_get_i64` (*nvs_handle_t* handle, **const** char *key, int64_t *out_value)
get int64_t value for given key

This function is the same as `nvs_get_i8` except for the data type.

`esp_err_t nvs_get_u64` (*nvs_handle_t* handle, **const** char *key, uint64_t *out_value)
get uint64_t value for given key

This function is the same as `nvs_get_i8` except for the data type.

`esp_err_t nvs_get_str` (*nvs_handle_t* handle, **const** char *key, char *out_value, size_t *length)
get string value for given key

These functions retrieve the data of an entry, given its key. If key does not exist, or the requested variable type doesn't match the type which was used when setting a value, an error is returned.

In case of any error, `out_value` is not modified.

All functions expect `out_value` to be a pointer to an already allocated variable of the given type.

`nvs_get_str` and `nvs_get_blob` functions support WinAPI-style length queries. To get the size necessary to store the value, call `nvs_get_str` or `nvs_get_blob` with zero `out_value` and non-zero pointer to length. Variable pointed to by length argument will be set to the required length. For `nvs_get_str`, this length includes the zero terminator. When calling `nvs_get_str` and `nvs_get_blob` with non-zero `out_value`, length has to be non-zero and has to point to the length available in `out_value`. It is suggested that `nvs_get/set_str` is used for zero-terminated C strings, and `nvs_get/set_blob` used for arbitrary data structures.

```
// Example (without error checking) of using nvs_get_str to get a string into
↳dynamic array:
size_t required_size;
nvs_get_str(my_handle, "server_name", NULL, &required_size);
char* server_name = malloc(required_size);
nvs_get_str(my_handle, "server_name", server_name, &required_size);

// Example (without error checking) of using nvs_get_blob to get a binary data
into a static array:
uint8_t mac_addr[6];
size_t size = sizeof(mac_addr);
nvs_get_blob(my_handle, "dst_mac_addr", mac_addr, &size);
```

Return

- ESP_OK if the value was retrieved successfully
- ESP_ERR_NVS_NOT_FOUND if the requested key doesn't exist
- ESP_ERR_NVS_INVALID_HANDLE if handle has been closed or is NULL
- ESP_ERR_NVS_INVALID_NAME if key name doesn't satisfy constraints
- ESP_ERR_NVS_INVALID_LENGTH if length is not sufficient to store data

Parameters

- [in] handle: Handle obtained from nvs_open function.
- [in] key: Key name. Maximal length is (NVS_KEY_NAME_MAX_SIZE-1) characters. Shouldn't be empty.
- [out] out_value: Pointer to the output value. May be NULL for nvs_get_str and nvs_get_blob, in this case required length will be returned in length argument.
- [inout] length: A non-zero pointer to the variable holding the length of out_value. In case out_value a zero, will be set to the length required to hold the value. In case out_value is not zero, will be set to the actual length of the value written. For nvs_get_str this includes zero terminator.

esp_err_t nvs_get_blob(*nvs_handle_t* handle, const char *key, void *out_value, size_t *length)
get blob value for given key

This function behaves the same as nvs_get_str, except for the data type.

esp_err_t nvs_open(const char *name, *nvs_open_mode_t* open_mode, *nvs_handle_t* *out_handle)
Open non-volatile storage with a given namespace from the default NVS partition.

Multiple internal ESP-IDF and third party application modules can store their key-value pairs in the NVS module. In order to reduce possible conflicts on key names, each module can use its own namespace. The default NVS partition is the one that is labelled "nvs" in the partition table.

Return

- ESP_OK if storage handle was opened successfully
- ESP_ERR_NVS_NOT_INITIALIZED if the storage driver is not initialized
- ESP_ERR_NVS_PART_NOT_FOUND if the partition with label "nvs" is not found
- ESP_ERR_NVS_NOT_FOUND id namespace doesn't exist yet and mode is NVS_READONLY
- ESP_ERR_NVS_INVALID_NAME if namespace name doesn't satisfy constraints
- other error codes from the underlying storage driver

Parameters

- [in] name: Namespace name. Maximal length is determined by the underlying implementation, but is guaranteed to be at least 15 characters. Shouldn't be empty.
- [in] open_mode: NVS_READWRITE or NVS_READONLY. If NVS_READONLY, will open a handle for reading only. All write requests will be rejected for this handle.
- [out] out_handle: If successful (return code is zero), handle will be returned in this argument.

esp_err_t nvs_open_from_partition(const char *part_name, const char *name, *nvs_open_mode_t* open_mode, *nvs_handle_t* *out_handle)
Open non-volatile storage with a given namespace from specified partition.

The behaviour is same as nvs_open() API. However this API can operate on a specified NVS partition instead of default NVS partition. Note that the specified partition must be registered with NVS using nvs_flash_init_partition() API.

Return

- ESP_OK if storage handle was opened successfully
- ESP_ERR_NVS_NOT_INITIALIZED if the storage driver is not initialized
- ESP_ERR_NVS_PART_NOT_FOUND if the partition with specified name is not found
- ESP_ERR_NVS_NOT_FOUND id namespace doesn't exist yet and mode is NVS_READONLY
- ESP_ERR_NVS_INVALID_NAME if namespace name doesn't satisfy constraints
- other error codes from the underlying storage driver

Parameters

- [in] part_name: Label (name) of the partition of interest for object read/write/erase
- [in] name: Namespace name. Maximal length is determined by the underlying implementation, but is guaranteed to be at least 15 characters. Shouldn't be empty.
- [in] open_mode: NVS_READWRITE or NVS_READONLY. If NVS_READONLY, will open a handle for reading only. All write requests will be rejected for this handle.

- [out] `out_handle`: If successful (return code is zero), handle will be returned in this argument.

`esp_err_t nvs_set_blob(nvs_handle_t handle, const char *key, const void *value, size_t length)`
set variable length binary value for given key

This family of functions set value for the key, given its name. Note that actual storage will not be updated until `nvs_commit` function is called.

Return

- `ESP_OK` if value was set successfully
- `ESP_ERR_NVS_INVALID_HANDLE` if handle has been closed or is `NULL`
- `ESP_ERR_NVS_READ_ONLY` if storage handle was opened as read only
- `ESP_ERR_NVS_INVALID_NAME` if key name doesn't satisfy constraints
- `ESP_ERR_NVS_NOT_ENOUGH_SPACE` if there is not enough space in the underlying storage to save the value
- `ESP_ERR_NVS_REMOVE_FAILED` if the value wasn't updated because flash write operation has failed. The value was written however, and update will be finished after re-initialization of `nvs`, provided that flash operation doesn't fail again.
- `ESP_ERR_NVS_VALUE_TOO_LONG` if the value is too long

Parameters

- [in] `handle`: Handle obtained from `nvs_open` function. Handles that were opened read only cannot be used.
- [in] `key`: Key name. Maximal length is 15 characters. Shouldn't be empty.
- [in] `value`: The value to set.
- [in] `length`: length of binary value to set, in bytes; Maximum length is 508000 bytes or (97.6% of the partition size - 4000) bytes whichever is lower.

`esp_err_t nvs_erase_key(nvs_handle_t handle, const char *key)`
Erase key-value pair with given key name.

Note that actual storage may not be updated until `nvs_commit` function is called.

Return

- `ESP_OK` if erase operation was successful
- `ESP_ERR_NVS_INVALID_HANDLE` if handle has been closed or is `NULL`
- `ESP_ERR_NVS_READ_ONLY` if handle was opened as read only
- `ESP_ERR_NVS_NOT_FOUND` if the requested key doesn't exist
- other error codes from the underlying storage driver

Parameters

- [in] `handle`: Storage handle obtained with `nvs_open`. Handles that were opened read only cannot be used.
- [in] `key`: Key name. Maximal length is determined by the underlying implementation, but is guaranteed to be at least 15 characters. Shouldn't be empty.

`esp_err_t nvs_erase_all(nvs_handle_t handle)`
Erase all key-value pairs in a namespace.

Note that actual storage may not be updated until `nvs_commit` function is called.

Return

- `ESP_OK` if erase operation was successful
- `ESP_ERR_NVS_INVALID_HANDLE` if handle has been closed or is `NULL`
- `ESP_ERR_NVS_READ_ONLY` if handle was opened as read only
- other error codes from the underlying storage driver

Parameters

- [in] `handle`: Storage handle obtained with `nvs_open`. Handles that were opened read only cannot be used.

`esp_err_t nvs_commit(nvs_handle_t handle)`
Write any pending changes to non-volatile storage.

After setting any values, `nvs_commit()` must be called to ensure changes are written to non-volatile storage. Individual implementations may write to storage at other times, but this is not guaranteed.

Return

- ESP_OK if the changes have been written successfully
- ESP_ERR_NVS_INVALID_HANDLE if handle has been closed or is NULL
- other error codes from the underlying storage driver

Parameters

- [in] handle: Storage handle obtained with nvs_open. Handles that were opened read only cannot be used.

void **nvs_close** (*nvs_handle_t* handle)

Close the storage handle and free any allocated resources.

This function should be called for each handle opened with nvs_open once the handle is not in use any more. Closing the handle may not automatically write the changes to nonvolatile storage. This has to be done explicitly using nvs_commit function. Once this function is called on a handle, the handle should no longer be used.

Parameters

- [in] handle: Storage handle to close

esp_err_t **nvs_get_stats** (**const** char *part_name, *nvs_stats_t* *nvs_stats)

Fill structure *nvs_stats_t*. It provides info about used memory the partition.

This function calculates to runtime the number of used entries, free entries, total entries, and amount namespace in partition.

```
// Example of nvs_get_stats() to get the number of used entries and free_
↳entries:
nvs_stats_t nvs_stats;
nvs_get_stats(NULL, &nvs_stats);
printf("Count: UsedEntries = (%d), FreeEntries = (%d), AllEntries = (%d)\n",
      nvs_stats.used_entries, nvs_stats.free_entries, nvs_stats.total_
↳entries);
```

Return

- ESP_OK if the changes have been written successfully. Return param nvs_stats will be filled.
- ESP_ERR_NVS_PART_NOT_FOUND if the partition with label "name" is not found. Return param nvs_stats will be filled 0.
- ESP_ERR_NVS_NOT_INITIALIZED if the storage driver is not initialized. Return param nvs_stats will be filled 0.
- ESP_ERR_INVALID_ARG if nvs_stats equal to NULL.
- ESP_ERR_INVALID_STATE if there is page with the status of INVALID. Return param nvs_stats will be filled not with correct values because not all pages will be counted. Counting will be interrupted at the first INVALID page.

Parameters

- [in] part_name: Partition name NVS in the partition table. If pass a NULL than will use NVS_DEFAULT_PART_NAME ("nvs").
- [out] nvs_stats: Returns filled structure nvs_states_t. It provides info about used memory the partition.

esp_err_t **nvs_get_used_entry_count** (*nvs_handle_t* handle, size_t *used_entries)

Calculate all entries in a namespace.

An entry represents the smallest storage unit in NVS. Strings and blobs may occupy more than one entry. Note that to find out the total number of entries occupied by the namespace, add one to the returned value used_entries (if err is equal to ESP_OK). Because the name space entry takes one entry.

```
// Example of nvs_get_used_entry_count() to get amount of all key-value pairs_
↳in one namespace:
nvs_handle_t handle;
nvs_open("namespace1", NVS_READWRITE, &handle);
...
size_t used_entries;
size_t total_entries_namespace;
```

(continues on next page)

(continued from previous page)

```

if(nvs_get_used_entry_count(handle, &used_entries) == ESP_OK){
    // the total number of entries occupied by the namespace
    total_entries_namespace = used_entries + 1;
}

```

Return

- ESP_OK if the changes have been written successfully. Return param used_entries will be filled valid value.
- ESP_ERR_NVS_NOT_INITIALIZED if the storage driver is not initialized. Return param used_entries will be filled 0.
- ESP_ERR_NVS_INVALID_HANDLE if handle has been closed or is NULL. Return param used_entries will be filled 0.
- ESP_ERR_INVALID_ARG if used_entries equal to NULL.
- Other error codes from the underlying storage driver. Return param used_entries will be filled 0.

Parameters

- [in] handle: Handle obtained from nvs_open function.
- [out] used_entries: Returns amount of used entries from a namespace.

nvs_iterator_t **nvs_entry_find**(const char *part_name, const char *namespace_name, *nvs_type_t* type)

Create an iterator to enumerate NVS entries based on one or more parameters.

```

// Example of listing all the key-value pairs of any type under specified_
↳partition and namespace
nvs_iterator_t it = nvs_entry_find(partition, namespace, NVS_TYPE_ANY);
while (it != NULL) {
    nvs_entry_info_t info;
    nvs_entry_info(it, &info);
    it = nvs_entry_next(it);
    printf("key '%s', type '%d' \n", info.key, info.type);
};
// Note: no need to release iterator obtained from nvs_entry_find function when
//       nvs_entry_find or nvs_entry_next function return NULL, indicating no_
↳other
//       element for specified criteria was found.
}

```

Return Iterator used to enumerate all the entries found, or NULL if no entry satisfying criteria was found. Iterator obtained through this function has to be released using nvs_release_iterator when not used any more.

Parameters

- [in] part_name: Partition name
- [in] namespace_name: Set this value if looking for entries with a specific namespace. Pass NULL otherwise.
- [in] type: One of nvs_type_t values.

nvs_iterator_t **nvs_entry_next** (*nvs_iterator_t* iterator)

Returns next item matching the iterator criteria, NULL if no such item exists.

Note that any copies of the iterator will be invalid after this call.

Return NULL if no entry was found, valid nvs_iterator_t otherwise.

Parameters

- [in] iterator: Iterator obtained from nvs_entry_find function. Must be non-NULL.

void **nvs_entry_info** (*nvs_iterator_t* iterator, *nvs_entry_info_t* *out_info)

Fills *nvs_entry_info_t* structure with information about entry pointed to by the iterator.

Parameters

- [in] iterator: Iterator obtained from nvs_entry_find or nvs_entry_next function. Must be non-NULL.
- [out] out_info: Structure to which entry information is copied.

void **nvs_release_iterator** (*nvs_iterator_t* iterator)

Release iterator.

Parameters

- [in] *iterator*: Release iterator obtained from `nvs_entry_find` function. NULL argument is allowed.

Structures

struct nvs_entry_info_t

information about entry obtained from `nvs_entry_info` function

Public Members

char **namespace_name**[16]

Namespace to which key-value belong

char **key**[16]

Key of stored key-value pair

nvs_type_t **type**

Type of stored key-value pair

struct nvs_stats_t

Note Info about storage space NVS.

Public Members

size_t **used_entries**

Amount of used entries.

size_t **free_entries**

Amount of free entries.

size_t **total_entries**

Amount all available entries.

size_t **namespace_count**

Amount name space.

Macros

ESP_ERR_NVS_BASE

Starting number of error codes

ESP_ERR_NVS_NOT_INITIALIZED

The storage driver is not initialized

ESP_ERR_NVS_NOT_FOUND

Id namespace doesn't exist yet and mode is NVS_READONLY

ESP_ERR_NVS_TYPE_MISMATCH

The type of set or get operation doesn't match the type of value stored in NVS

ESP_ERR_NVS_READ_ONLY

Storage handle was opened as read only

ESP_ERR_NVS_NOT_ENOUGH_SPACE

There is not enough space in the underlying storage to save the value

ESP_ERR_NVS_INVALID_NAME

Namespace name doesn't satisfy constraints

ESP_ERR_NVS_INVALID_HANDLE

Handle has been closed or is NULL

ESP_ERR_NVS_REMOVE_FAILED

The value wasn't updated because flash write operation has failed. The value was written however, and update will be finished after re-initialization of nvs, provided that flash operation doesn't fail again.

ESP_ERR_NVS_KEY_TOO_LONG

Key name is too long

ESP_ERR_NVS_PAGE_FULL

Internal error; never returned by nvs API functions

ESP_ERR_NVS_INVALID_STATE

NVS is in an inconsistent state due to a previous error. Call `nvs_flash_init` and `nvs_open` again, then retry.

ESP_ERR_NVS_INVALID_LENGTH

String or blob length is not sufficient to store data

ESP_ERR_NVS_NO_FREE_PAGES

NVS partition doesn't contain any empty pages. This may happen if NVS partition was truncated. Erase the whole partition and call `nvs_flash_init` again.

ESP_ERR_NVS_VALUE_TOO_LONG

String or blob length is longer than supported by the implementation

ESP_ERR_NVS_PART_NOT_FOUND

Partition with specified name is not found in the partition table

ESP_ERR_NVS_NEW_VERSION_FOUND

NVS partition contains data in new format and cannot be recognized by this version of code

ESP_ERR_NVS_XTS_ENCR_FAILED

XTS encryption failed while writing NVS entry

ESP_ERR_NVS_XTS_DECR_FAILED

XTS decryption failed while reading NVS entry

ESP_ERR_NVS_XTS_CFG_FAILED

XTS configuration setting failed

ESP_ERR_NVS_XTS_CFG_NOT_FOUND

XTS configuration not found

ESP_ERR_NVS_ENCR_NOT_SUPPORTED

NVS encryption is not supported in this version

ESP_ERR_NVS_KEYS_NOT_INITIALIZED

NVS key partition is uninitialized

ESP_ERR_NVS_CORRUPT_KEY_PART

NVS key partition is corrupt

ESP_ERR_NVS_CONTENT_DIFFERS

Internal error; never returned by nvs API functions. NVS key is different in comparison

NVS_DEFAULT_PART_NAME

Default partition name of the NVS partition in the partition table

NVS_PART_NAME_MAX_SIZE

maximum length of partition name (excluding null terminator)

Type Definitions

```
typedef uint32_t nvs_handle_t
```

Opaque pointer type representing non-volatile storage handle

```
typedef nvs_handle_t nvs_handle
```

```
typedef nvs_open_mode_t nvs_open_mode
```

```
typedef struct nvs_opaque_iterator_t *nvs_iterator_t  
    Opaque pointer type representing iterator to nvs entries
```

Enumerations

```
enum nvs_open_mode_t  
    Mode of opening the non-volatile storage.
```

Values:

```
NVS_READONLY
```

Read only

```
NVS_READWRITE
```

Read and write

```
enum nvs_type_t
```

Types of variables.

Values:

```
NVS_TYPE_U8 = 0x01
```

Type uint8_t

```
NVS_TYPE_I8 = 0x11
```

Type int8_t

```
NVS_TYPE_U16 = 0x02
```

Type uint16_t

```
NVS_TYPE_I16 = 0x12
```

Type int16_t

```
NVS_TYPE_U32 = 0x04
```

Type uint32_t

```
NVS_TYPE_I32 = 0x14
```

Type int32_t

```
NVS_TYPE_U64 = 0x08
```

Type uint64_t

```
NVS_TYPE_I64 = 0x18
```

Type int64_t

```
NVS_TYPE_STR = 0x21
```

Type string

```
NVS_TYPE_BLOB = 0x42
```

Type blob

```
NVS_TYPE_ANY = 0xff
```

Must be last

2.6.4 NVS Partition Generator Utility

Introduction

The utility `nvs_flash/nvs_partition_generator/nvs_partition_gen.py` creates a binary file based on key-value pairs provided in a CSV file. The binary file is compatible with NVS architecture defined in *Non-Volatile Storage*. This utility is ideally suited for generating a binary blob, containing data specific to ODM/OEM, which can be flashed externally at the time of device manufacturing. This allows manufacturers to generate many instances of the same application firmware with customized parameters for each device, such as a serial number.

Prerequisites

To use this utility in encryption mode, install the following packages:

- cryptography package

All the required packages are included in *requirements.txt* in the root of the esp-idf directory.

CSV file format

Each line of a .csv file should contain 4 parameters, separated by a comma. The table below provides the description for each of these parameters.

| No. | Parameter | Description | Notes |
|-----|-----------|--|---|
| 1 | Key | Key of the data. The data can be accessed later from an application using this key. | |
| 2 | Type | Supported values are <code>file</code> , <code>data</code> and <code>namespace</code> . | |
| 3 | Encoding | Supported values are: <code>u8</code> , <code>i8</code> , <code>u16</code> , <code>u32</code> , <code>i32</code> , <code>string</code> , <code>hex2bin</code> , <code>base64</code> and <code>binary</code> . This specifies how actual data values are encoded in the resulting binary file. The difference between the <code>string</code> and <code>binary</code> encoding is that <code>string</code> data is terminated with a NULL character, whereas <code>binary</code> data is not. | As of now, for the <code>file</code> type, only <code>hex2bin</code> , <code>base64</code> , <code>string</code> , and <code>binary</code> encoding is supported. |
| 4 | Value | Data value. | Encoding and Value cells for the <code>namespace</code> field type should be empty. Encoding and Value of <code>namespace</code> is fixed and is not configurable. Any values in these cells are ignored. |

Note: The first line of the CSV file should always be the column header and it is not configurable.

Below is an example dump of such a CSV file:

```
key,type,encoding,value      <-- column header
namespace_name,namespace,,  <-- First entry should be of type "namespace"
key1,data,u8,1
key2,file,string,/path/to/file
```

Note:

Make sure there are no spaces:

- before and after `'`,
 - at the end of each line in a CSV file
-

NVS Entry and Namespace association

When a namespace entry is encountered in a CSV file, each following entry will be treated as part of that namespace until the next namespace entry is found. At this point, all the following entries will be treated as part of the new namespace.

Note: First entry in a CSV file should always be a namespace entry.

Multipage Blob Support

By default, binary blobs are allowed to span over multiple pages and are written in the format mentioned in Section *Structure of entry*. If you intend to use an older format, the utility provides an option to disable this feature.

Encryption Support

The NVS Partition Generator utility also allows you to create an encrypted binary file. The utility uses the AES-XTS encryption. Please refer to *NVS Encryption* for more details.

Decryption Support

This utility allows you to decrypt an encrypted NVS binary file. The utility uses an NVS binary file encrypted using AES-XTS encryption. Please refer to *NVS Encryption* for more details.

Running the utility

Usage:

```
python nvs_partition_gen.py [-h] {generate,generate-key,encrypt,decrypt} ...
```

Optional Arguments:

| No. | Parameter | Description |
|-----|------------|--|
| 1 | -h, --help | show this help message and exit |

Commands:

Run `nvs_partition_gen.py {command} -h` **for** additional help

| No. | Parameter | Description |
|-----|--------------|-------------------------------------|
| 1 | generate | Generate NVS partition |
| 2 | generate-key | Generate keys for encryption |
| 3 | encrypt | Generate NVS encrypted partition |

(continues on next page)

(continued from previous page)

| | | | |
|---------|---------|---------------------------------|---|
| 4 | decrypt | Decrypt NVS encrypted partition | |
| ↩ | | | ↪ |
| +-----+ | | | |
| ↩ | ----- | | ↪ |

To generate NVS partition (Default):**Usage:**

```
python nvs_partition_gen.py generate [-h] [--version {1,2}] [--outdir OUTDIR]
                                     input output size
```

Positional Arguments:

| Parameter | Description |
|-----------|--|
| input | Path to CSV file to parse |
| output | Path to output NVS binary file |
| size | Size of NVS partition in bytes (must be multiple of 4096) |

Optional Arguments:

| Parameter | Description |
|-----------------|--|
| -h, --help | show this help message and exit |
| --version {1,2} | Set multipage blob version. Version 1 - Multipage blob support disabled. Version 2 - Multipage blob support enabled. Default: Version 2 |
| --outdir OUTDIR | Output directory to store files created (Default: current directory) |

You can run the utility to generate NVS partition using the command below: A sample CSV file is provided with the utility:

```
python nvs_partition_gen.py generate sample_singlepage_blob.csv sample.bin 0x3000
```

To generate only encryption keys:

Usage:

```
python nvs_partition_gen.py generate-key [-h] [--keyfile KEYFILE]
                                         [--outdir OUTDIR]
```

Optional Arguments:

| Parameter | Description |
|-------------------|--|
| -h, --help | show this help message and exit |
| --keyfile KEYFILE | Path to output encryption keys file |
| --outdir OUTDIR | Output directory to store files created. (Default: current directory) |

You can run the utility to generate only encryption keys using the command below:

```
python nvs_partition_gen.py generate-key
```

To generate encrypted NVS partition:

Usage:

```
python nvs_partition_gen.py encrypt [-h] [--version {1,2}] [--keygen]
                                     [--keyfile KEYFILE] [--inputkey_
                                     ↪ INPUTKEY]
                                     [--outdir OUTDIR]
                                     input output size
```

Positional Arguments:

| Parameter | Description |
|-----------|--------------------------------|
| input | Path to CSV file to parse |
| output | Path to output NVS binary file |

(continues on next page)

(continued from previous page)

| Parameter | Description |
|---------------------|--|
| size | Size of NVS partition in bytes (must be multiple of 4096) |
| Optional Arguments: | |
| -h, --help | show this help message and exit |
| --version {1,2} | Set multipage blob version. Version 1 - Multipage blob support disabled. Version 2 - Multipage blob support enabled. Default: Version 2 |
| --keygen | Generates key for encrypting NVS partition |
| --keyfile KEYFILE | Path to output encryption keys file |
| --inputkey INPUTKEY | File having key for encrypting NVS partition |
| --outdir OUTDIR | Output directory to store files created (Default: current directory) |

You can run the utility to encrypt NVS partition using the command below: A sample CSV file is provided with the utility:

- Encrypt by allowing the utility to generate encryption keys:

```
python nvs_partition_gen.py encrypt sample_singlepage_blob.csv sample_encr.bin
0x3000 --keygen
```

Note: Encryption key of the following format <outdir>/keys/keys-<timestamp>.bin is created.

- Encrypt by allowing the utility to generate encryption keys and store it in provided custom filename:

```
python nvs_partition_gen.py encrypt sample_singlepage_blob.csv sample_encr.bin
↪0x3000 --keygen --keyfile sample_keys.bin
```

Note: Encryption key of the following format <outdir>/keys/sample_keys.bin is created.

Note: This newly created file having encryption keys in keys/ directory is compatible with NVS key-partition structure. Refer to *NVS key partition* for more details.

- Encrypt by providing the encryption keys as input binary file:

```
python nvs_partition_gen.py encrypt sample_singlepage_blob.csv sample_encr.bin
↪0x3000 --inputkey sample_keys.bin
```

To decrypt encrypted NVS partition:

Usage:

```
python nvs_partition_gen.py decrypt [-h] [--outdir OUTDIR] input key_
↪output
```

Positional Arguments:

| Parameter | Description |
|-----------|--|
| input | Path to encrypted NVS partition file to parse |
| key | Path to file having keys for decryption |
| output | Path to output decrypted binary file |

Optional Arguments:

| Parameter | Description |
|-----------------|---|
| -h, --help | show this help message and exit |
| --outdir OUTDIR | Output directory to store files created (Default: current directory) |

You can run the utility to decrypt encrypted NVS partition using the command below:

```
python nvs_partition_gen.py decrypt sample_encr.bin sample_keys.bin sample_decr.bin
```

You can also provide the format version number:

- Multipage Blob Support Disabled (Version 1)
- Multipage Blob Support Enabled (Version 2)

Multipage Blob Support Disabled (Version 1): You can run the utility in this format by setting the version parameter to 1, as shown below. A sample CSV file is provided with the utility:

```
python nvs_partition_gen.py generate sample_singlepage_blob.csv sample.bin 0x3000 --  
↪-version 1
```

Multipage Blob Support Enabled (Version 2): You can run the utility in this format by setting the version parameter to 2, as shown below. A sample CSV file is provided with the utility:

```
python nvs_partition_gen.py generate sample_multipage_blob.csv sample.bin 0x4000 --  
↪version 2
```

Note: *Minimum NVS Partition Size needed is 0x3000 bytes.*

Note: *When flashing the binary onto the device, make sure it is consistent with the application's sdkconfig.*

Caveats

- Utility does not check for duplicate keys and will write data pertaining to both keys. You need to make sure that the keys are distinct.
- Once a new page is created, no data will be written in the space left on the previous page. Fields in the CSV file need to be ordered in such a way as to optimize memory.
- 64-bit datatype is not yet supported.

2.6.5 SD/SDIO/MMC Driver

Overview

The SD/SDIO/MMC driver currently supports SD memory, SDIO cards, and eMMC chips. This is a protocol level driver built on top of SDMMC and SD SPI host drivers.

SDMMC and SD SPI host drivers ([driver/include/driver/sdmmc_host.h](#)) provide API functions for:

- Sending commands to slave devices
- Sending and receiving data
- Handling error conditions within the bus

For functions used to initialize and configure:

- SDMMC host, see [SDMMC Host API](#)
- SD SPI host, see [SD SPI Host API](#)

The SDMMC protocol layer described in this document handles the specifics of the SD protocol, such as the card initialization and data transfer commands.

The protocol layer works with the host via the `sdmmc_host_t` structure. This structure contains pointers to various functions of the host.

Application Example

An example which combines the SDMMC driver with the FATFS library is provided in the [storage/sd_card](#) directory of ESP-IDF examples. This example initializes the card, then writes and reads data from it using POSIX and C library APIs. See README.md file in the example directory for more information.

Protocol layer API

The protocol layer is given the `sdmmc_host_t` structure. This structure describes the SD/MMC host driver, lists its capabilities, and provides pointers to functions of the driver. The protocol layer stores card-specific information in the `sdmmc_card_t` structure. When sending commands to the SD/MMC host driver, the protocol layer uses the `sdmmc_command_t` structure to describe the command, arguments, expected return values, and data to transfer if there is any.

Using API with SD memory cards

1. To initialize the host, call the host driver functions, e.g., `sdmmc_host_init()`, `sdmmc_host_init_slot()`.
2. To initialize the card, call `sdmmc_card_init()` and pass to it the parameters `host` - the host driver information, and `card` - a pointer to the structure `sdmmc_card_t` which will be filled with information about the card when the function completes.
3. To read and write sectors of the card, use `sdmmc_read_sectors()` and `sdmmc_write_sectors()` respectively and pass to it the parameter `card` - a pointer to the card information structure.
4. If the card is not used anymore, call the host driver function - e.g., `sdmmc_host_deinit()` - to disable the host peripheral and free the resources allocated by the driver.

Using API with eMMC chips From the protocol layer's perspective, eMMC memory chips behave exactly like SD memory cards. Even though eMMCs are chips and do not have a card form factor, the terminology for SD cards can still be applied to eMMC due to the similarity of the protocol (`sdmmc_card_t`, `sdmmc_card_init`). Note that eMMC chips cannot be used over SPI, which makes them incompatible with the SD SPI host driver.

To initialize eMMC memory and perform read/write operations, follow the steps listed for SD cards in the previous section.

Using API with SDIO cards Initialization and the probing process is the same as with SD memory cards. The only difference is in data transfer commands in SDIO mode.

During the card initialization and probing, performed with `sdmmc_card_init()`, the driver only configures the following registers of the IO card:

1. The IO portion of the card is reset by setting RES bit in the I/O Abort (0x06) register.
2. If 4-line mode is enabled in host and slot configuration, the driver attempts to set the Bus width field in the Bus Interface Control (0x07) register. If setting the field is successful, which means that the slave supports 4-line mode, the host is also switched to 4-line mode.
3. If high-speed mode is enabled in the host configuration, the SHS bit is set in the High Speed (0x13) register.

In particular, the driver does not set any bits in (1) I/O Enable and Int Enable registers, (2) I/O block sizes, etc. Applications can set them by calling `sdmmc_io_write_byte()`.

For card configuration and data transfer, choose the pair of functions relevant to your case from the table below.

| Action | Read Function | Write Function |
|--|-------------------------------------|--------------------------------------|
| Read and write a single byte using IO_RW_DIRECT (CMD52) | <code>sdmmc_io_read_byte()</code> | <code>sdmmc_io_write_byte()</code> |
| Read and write multiple bytes using IO_RW_EXTENDED (CMD53) in byte mode | <code>sdmmc_io_read_bytes()</code> | <code>sdmmc_io_write_bytes()</code> |
| Read and write blocks of data using IO_RW_EXTENDED (CMD53) in block mode | <code>sdmmc_io_read_blocks()</code> | <code>sdmmc_io_write_blocks()</code> |

SDIO interrupts can be enabled by the application using the function `sdmmc_io_enable_int()`. When using SDIO in 1-line mode, the D1 line also needs to be connected to use SDIO interrupts.

If you want the application to wait until the SDIO interrupt occurs, use `sdmmc_io_wait_int()`.

There is a component ESSL (ESP Serial Slave Link) to use if you are communicating with an ESP32 SDIO slave. See [ESP Serial Slave Link](#) and example [peripherals/sdio/host](#).

Combo (memory + IO) cards The driver does not support SD combo cards. Combo cards are treated as IO cards.

Thread safety Most applications need to use the protocol layer only in one task. For this reason, the protocol layer does not implement any kind of locking on the `sdmmc_card_t` structure, or when accessing SDMMC or SD SPI host drivers. Such locking is usually implemented on a higher layer, e.g., in the filesystem driver.

API Reference

Header File

- `sdmmc/include/sdmmc_cmd.h`

Functions

`esp_err_t sdmmc_card_init (const sdmmc_host_t *host, sdmmc_card_t *out_card)`

Probe and initialize SD/MMC card using given host

Note Only SD cards (SDSC and SDHC/SDXC) are supported now. Support for MMC/eMMC cards will be added later.

Return

- ESP_OK on success
- One of the error codes from SDMMC host controller

Parameters

- `host`: pointer to structure defining host controller
- `out_card`: pointer to structure which will receive information about the card when the function completes

void `sdmmc_card_print_info` (FILE *stream, const sdmmc_card_t *card)

Print information about the card to a stream.

Parameters

- `stream`: stream obtained using `fopen` or `fdopen`
- `card`: card information structure initialized using `sdmmc_card_init`

`esp_err_t sdmmc_write_sectors (sdmmc_card_t *card, const void *src, size_t start_sector, size_t sector_count)`

Write given number of sectors to SD/MMC card

Return

- ESP_OK on success
- One of the error codes from SDMMC host controller

Parameters

- `card`: pointer to card information structure previously initialized using `sdmmc_card_init`
- `src`: pointer to data buffer to read data from; data size must be equal to `sector_count * card->csd.sector_size`
- `start_sector`: sector where to start writing
- `sector_count`: number of sectors to write

`esp_err_t sdmmc_read_sectors (sdmmc_card_t *card, void *dst, size_t start_sector, size_t sector_count)`

Write given number of sectors to SD/MMC card

Return

- ESP_OK on success

- One of the error codes from SDMMC host controller

Parameters

- `card`: pointer to card information structure previously initialized using `sdmmc_card_init`
- `dst`: pointer to data buffer to write into; buffer size must be at least `sector_count * card->csd.sector_size`
- `start_sector`: sector where to start reading
- `sector_count`: number of sectors to read

esp_err_t `sdmmc_io_read_byte` (*sdmmc_card_t* *`card`, *uint32_t* `function`, *uint32_t* `reg`, *uint8_t* *`out_byte`)

Read one byte from an SDIO card using `IO_RW_DIRECT` (CMD52)

Return

- `ESP_OK` on success
- One of the error codes from SDMMC host controller

Parameters

- `card`: pointer to card information structure previously initialized using `sdmmc_card_init`
- `function`: IO function number
- `reg`: byte address within IO function
- [`out`] `out_byte`: output, receives the value read from the card

esp_err_t `sdmmc_io_write_byte` (*sdmmc_card_t* *`card`, *uint32_t* `function`, *uint32_t* `reg`, *uint8_t* `in_byte`, *uint8_t* *`out_byte`)

Write one byte to an SDIO card using `IO_RW_DIRECT` (CMD52)

Return

- `ESP_OK` on success
- One of the error codes from SDMMC host controller

Parameters

- `card`: pointer to card information structure previously initialized using `sdmmc_card_init`
- `function`: IO function number
- `reg`: byte address within IO function
- `in_byte`: value to be written
- [`out`] `out_byte`: if not `NULL`, receives new byte value read from the card (read-after-write).

esp_err_t `sdmmc_io_read_bytes` (*sdmmc_card_t* *`card`, *uint32_t* `function`, *uint32_t* `addr`, *void* *`dst`, *size_t* `size`)

Read multiple bytes from an SDIO card using `IO_RW_EXTENDED` (CMD53)

This function performs read operation using `CMD53` in byte mode. For block mode, see `sdmmc_io_read_blocks`.

Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_SIZE` if `size` exceeds 512 bytes
- One of the error codes from SDMMC host controller

Parameters

- `card`: pointer to card information structure previously initialized using `sdmmc_card_init`
- `function`: IO function number
- `addr`: byte address within IO function where reading starts
- `dst`: buffer which receives the data read from card
- `size`: number of bytes to read

esp_err_t `sdmmc_io_write_bytes` (*sdmmc_card_t* *`card`, *uint32_t* `function`, *uint32_t* `addr`, *const* *void* *`src`, *size_t* `size`)

Write multiple bytes to an SDIO card using `IO_RW_EXTENDED` (CMD53)

This function performs write operation using `CMD53` in byte mode. For block mode, see `sdmmc_io_write_blocks`.

Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_SIZE` if `size` exceeds 512 bytes
- One of the error codes from SDMMC host controller

Parameters

- `card`: pointer to card information structure previously initialized using `sdmmc_card_init`
- `function`: IO function number
- `addr`: byte address within IO function where writing starts
- `src`: data to be written
- `size`: number of bytes to write

`esp_err_t sdmmc_io_read_blocks` (*`sdmmc_card_t *card`, `uint32_t function`, `uint32_t addr`, `void *dst`, `size_t size`)*

Read blocks of data from an SDIO card using IO_RW_EXTENDED (CMD53)

This function performs read operation using CMD53 in block mode. For byte mode, see `sdmmc_io_read_bytes`.

Return

- ESP_OK on success
- ESP_ERR_INVALID_SIZE if `size` is not divisible by 512 bytes
- One of the error codes from SDMMC host controller

Parameters

- `card`: pointer to card information structure previously initialized using `sdmmc_card_init`
- `function`: IO function number
- `addr`: byte address within IO function where writing starts
- `dst`: buffer which receives the data read from card
- `size`: number of bytes to read, must be divisible by the card block size.

`esp_err_t sdmmc_io_write_blocks` (*`sdmmc_card_t *card`, `uint32_t function`, `uint32_t addr`, `const void *src`, `size_t size`)*

Write blocks of data to an SDIO card using IO_RW_EXTENDED (CMD53)

This function performs write operation using CMD53 in block mode. For byte mode, see `sdmmc_io_write_bytes`.

Return

- ESP_OK on success
- ESP_ERR_INVALID_SIZE if `size` is not divisible by 512 bytes
- One of the error codes from SDMMC host controller

Parameters

- `card`: pointer to card information structure previously initialized using `sdmmc_card_init`
- `function`: IO function number
- `addr`: byte address within IO function where writing starts
- `src`: data to be written
- `size`: number of bytes to read, must be divisible by the card block size.

`esp_err_t sdmmc_io_enable_int` (*`sdmmc_card_t *card`)*

Enable SDIO interrupt in the SDMMC host

Return

- ESP_OK on success
- ESP_ERR_NOT_SUPPORTED if the host controller does not support IO interrupts

Parameters

- `card`: pointer to card information structure previously initialized using `sdmmc_card_init`

`esp_err_t sdmmc_io_wait_int` (*`sdmmc_card_t *card`, `TickType_t timeout_ticks`)*

Block until an SDIO interrupt is received

Slave uses D1 line to signal interrupt condition to the host. This function can be used to wait for the interrupt.

Return

- ESP_OK if the interrupt is received
- ESP_ERR_NOT_SUPPORTED if the host controller does not support IO interrupts
- ESP_ERR_TIMEOUT if the interrupt does not happen in `timeout_ticks`

Parameters

- `card`: pointer to card information structure previously initialized using `sdmmc_card_init`
- `timeout_ticks`: time to wait for the interrupt, in RTOS ticks

```
esp_err_t sdmmc_io_get_cis_data(sdmmc_card_t *card, uint8_t *out_buffer, size_t buffer_size,
                               size_t *inout_cis_size)
```

Get the data of CIS region of a SDIO card.

You may provide a buffer not sufficient to store all the CIS data. In this case, this functions store as much data into your buffer as possible. Also, this function will try to get and return the size required for you.

Return

- ESP_OK: on success
- ESP_ERR_INVALID_RESPONSE: if the card does not (correctly) support CIS.
- ESP_ERR_INVALID_SIZE: CIS_CODE_END found, but buffer_size is less than required size, which is stored in the inout_cis_size then.
- ESP_ERR_NOT_FOUND: if the CIS_CODE_END not found. Increase input value of inout_cis_size or set it to 0, if you still want to search for the end; output value of inout_cis_size is invalid in this case.
- and other error code return from sdmmc_io_read_bytes

Parameters

- card: pointer to card information structure previously initialized using sdmmc_card_init
- out_buffer: Output buffer of the CIS data
- buffer_size: Size of the buffer.
- inout_cis_size: Mandatory, pointer to a size, input and output.
 - input: Limitation of maximum searching range, should be 0 or larger than buffer_size. The function searches for CIS_CODE_END until this range. Set to 0 to search infinitely.
 - output: The size required to store all the CIS data, if CIS_CODE_END is found.

```
esp_err_t sdmmc_io_print_cis_info(uint8_t *buffer, size_t buffer_size, FILE *fp)
```

Parse and print the CIS information of a SDIO card.

Note Not all the CIS codes and all kinds of tuples are supported. If you see some unresolved code, you can add the parsing of these code in sdmmc_io.c and contribute to the IDF through the Github repository.

```
using sdmmc_card_init
```

Return

- ESP_OK: on success
- ESP_ERR_NOT_SUPPORTED: if the value from the card is not supported to be parsed.
- ESP_ERR_INVALID_SIZE: if the CIS size fields are not correct.

Parameters

- buffer: Buffer to parse
- buffer_size: Size of the buffer.
- fp: File pointer to print to, set to NULL to print to stdout.

Header File

- [driver/include/driver/sdmmc_types.h](#)

Structures

```
struct sdmmc_csd_t
```

Decoded values from SD card Card Specific Data register

Public Members

```
int csd_ver
```

CSD structure format

```
int mmc_ver
```

MMC version (for CID format)

```
int capacity
```

total number of sectors

`int sector_size`
sector size in bytes

`int read_block_len`
block length for reads

`int card_command_class`
Card Command Class for SD

`int tr_speed`
Max transfer speed

`struct sdmmc_cid_t`
Decoded values from SD card Card IDentification register

Public Members

`int mfg_id`
manufacturer identification number

`int oem_id`
OEM/product identification number

`char name[8]`
product name (MMC v1 has the longest)

`int revision`
product revision

`int serial`
product serial number

`int date`
manufacturing date

`struct sdmmc_scr_t`
Decoded values from SD Configuration Register

Public Members

`int sd_spec`
SD Physical layer specification version, reported by card

`int bus_width`
bus widths supported by card: BIT(0) —1-bit bus, BIT(2) —4-bit bus

`struct sdmmc_ext_csd_t`
Decoded values of Extended Card Specific Data

Public Members

`uint8_t power_class`
Power class used by the card

`struct sdmmc_switch_func_rsp_t`
SD SWITCH_FUNC response buffer

Public Members

`uint32_t data[512 / 8 / sizeof(uint32_t)]`
response data

struct sdmmc_command_t
SD/MMC command information

Public Members

uint32_t **opcode**
SD or MMC command index

uint32_t **arg**
SD/MMC command argument

sdmmc_response_t **response**
response buffer

void ***data**
buffer to send or read into

size_t **datalen**
length of data buffer

size_t **blklen**
block length

int **flags**
see below

esp_err_t **error**
error returned from transfer

int **timeout_ms**
response timeout, in milliseconds

struct sdmmc_host_t
SD/MMC Host description

This structure defines properties of SD/MMC host and functions of SD/MMC host which can be used by upper layers.

Public Members

uint32_t **flags**
flags defining host properties

int **slot**
slot number, to be passed to host functions

int **max_freq_khz**
max frequency supported by the host

float **io_voltage**
I/O voltage used by the controller (voltage switching is not supported)

esp_err_t (***init**) (void)
Host function to initialize the driver

esp_err_t (***set_bus_width**) (int slot, size_t width)
host function to set bus width

size_t (***get_bus_width**) (int slot)
host function to get bus width

esp_err_t (***set_bus_ddr_mode**) (int slot, bool ddr_enable)
host function to set DDR mode

esp_err_t (***set_card_clk**) (int slot, uint32_t freq_khz)
host function to set card clock frequency

esp_err_t (***do_transaction**) (int slot, *sdmmc_command_t* *cmdinfo)
host function to do a transaction

esp_err_t (***deinit**) (void)
host function to deinitialize the driver

esp_err_t (***io_int_enable**) (int slot)
Host function to enable SDIO interrupt line

esp_err_t (***io_int_wait**) (int slot, TickType_t timeout_ticks)
Host function to wait for SDIO interrupt line to be active

int **command_timeout_ms**
timeout, in milliseconds, of a single command. Set to 0 to use the default value.

struct sdmmc_card_t
SD/MMC card information structure

Public Members

sdmmc_host_t **host**
Host with which the card is associated

uint32_t **ocr**
OCR (Operation Conditions Register) value

sdmmc_cid_t **cid**
decoded CID (Card IDentification) register value

sdmmc_response_t **raw_cid**
raw CID of MMC card to be decoded after the CSD is fetched in the data transfer mode

sdmmc_csd_t **csd**
decoded CSD (Card-Specific Data) register value

sdmmc_scr_t **scr**
decoded SCR (SD card Configuration Register) value

sdmmc_ext_csd_t **ext_csd**
decoded EXT_CSD (Extended Card Specific Data) register value

uint16_t **rca**
RCA (Relative Card Address)

uint16_t **max_freq_khz**
Maximum frequency, in kHz, supported by the card

uint32_t **is_mem** : 1
Bit indicates if the card is a memory card

uint32_t **is_sdio** : 1
Bit indicates if the card is an IO card

uint32_t **is_mmc** : 1
Bit indicates if the card is MMC

uint32_t **num_io_functions** : 3
If **is_sdio** is 1, contains the number of IO functions on the card

uint32_t **log_bus_width** : 2
log₂(bus width supported by card)

uint32_t **is_ddr** : 1
Card supports DDR mode

uint32_t **reserved** : 23
Reserved for future expansion

Macros

SDMMC_HOST_FLAG_1BIT
host supports 1-line SD and MMC protocol

SDMMC_HOST_FLAG_4BIT
host supports 4-line SD and MMC protocol

SDMMC_HOST_FLAG_8BIT
host supports 8-line MMC protocol

SDMMC_HOST_FLAG_SPI
host supports SPI protocol

SDMMC_HOST_FLAG_DDR
host supports DDR mode for SD/MMC

SDMMC_FREQ_DEFAULT
SD/MMC Default speed (limited by clock divider)

SDMMC_FREQ_HIGHSPEED
SD High speed (limited by clock divider)

SDMMC_FREQ_PROBING
SD/MMC probing speed

SDMMC_FREQ_52M
MMC 52MHz speed

SDMMC_FREQ_26M
MMC 26MHz speed

Type Definitions

```
typedef uint32_t sdmmc_response_t[4]  
    SD/MMC command response buffer
```

2.6.6 SPI Flash API

Overview

The `spi_flash` component contains API functions related to reading, writing, erasing, memory mapping for data in the external flash. The `spi_flash` component also has higher-level API functions which work with partitions defined in the [partition table](#).

Different from the API before IDF v4.0, the functionality is not limited to the “main” SPI flash chip (the same SPI flash chip from which program runs). With different chip pointers, you can access to external flashes chips on not only SPI0/1 but also HSPI/VSPI buses.

Note: Flash APIs after IDF v4.0 are no longer *atomic*. A writing operation during another on-going read operation, on the overlapped flash address, may cause the return data from the read operation to be partly same as before, and partly updated as new written.

Kconfig option `CONFIG_SPI_FLASH_USE_LEGACY_IMPL` can be used to switch `spi_flash_*` functions back to the implementation before IDF v4.0. However, the code size may get bigger if you use the new API and the old API the same time.

Encrypted reads and writes use the old implementation, even if `CONFIG_SPI_FLASH_USE_LEGACY_IMPL` is not enabled. As such, encrypted flash operations are only supported with the main flash chip (and not with other flash chips on SPI1 with different CS).

Initializing a flash device

To use `esp_flash_*` APIs, you need to have a chip initialized on a certain SPI bus.

1. Call `spi_bus_initialize()` to properly initialize an SPI bus. This functions initialize the resources (I/O, DMA, interrupts) shared among devices attached to this bus.
2. Call `spi_bus_add_flash_device()` to attach the flash device onto the bus. This allocates memory, and fill the members for the `esp_flash_t` structure. The CS I/O is also initialized here.
3. Call `esp_flash_init()` to actually communicate with the chip. This will also detect the chip type, and influence the following operations.

Note: Multiple flash chips can be attached to the same bus now. However, using `esp_flash_*` devices and `spi_device_*` devices on the same SPI bus is not supported yet.

SPI flash access API

This is the set of API functions for working with data in flash:

- `esp_flash_read()` reads data from flash to RAM
- `esp_flash_write()` writes data from RAM to flash
- `esp_flash_erase_region()` erases specific region of flash
- `esp_flash_erase_chip()` erases the whole flash
- `esp_flash_get_chip_size()` returns flash chip size, in bytes, as configured in menuconfig

Generally, try to avoid using the raw SPI flash functions to the “main” SPI flash chip in favour of *partition-specific functions*.

SPI Flash Size

The SPI flash size is configured by writing a field in the software bootloader image header, flashed at offset 0x1000.

By default, the SPI flash size is detected by `esptool.py` when this bootloader is written to flash, and the header is updated with the correct size. Alternatively, it is possible to generate a fixed flash size by setting `CONFIG_ESPTOOLPY_FLASHSIZE` in project configuration.

If it is necessary to override the configured flash size at runtime, it is possible to set the `chip_size` member of the `g_rom_flashchip` structure. This size is used by `esp_flash_*` functions (in both software & ROM) to check the bounds.

Concurrency Constraints for flash on SPI1

Because the SPI1 flash is also used for firmware execution via the instruction & data caches, these caches must be disabled while reading/writing/erasing. This means that both CPUs must be running code from IRAM and must only be reading data from DRAM while flash write operations occur.

If you use the API functions documented here, then these constraints are applied automatically and transparently. However, note that it will have some performance impact on other tasks in the system.

There are no such constraints and impacts for flash chips on other SPI buses than SPI0/1.

For differences between IRAM, DRAM, and flash cache, please refer to the *application memory layout* documentation.

To avoid reading flash cache accidentally, when one CPU initiates a flash write or erase operation, the other CPU is put into a blocked state, and all non-IRAM-safe interrupts are disabled on both CPUs until the flash operation completes.

If one CPU initiates a flash write or erase operation, the other CPU is put into a blocked state to avoid reading flash cache accidentally. All interrupts not safe for IRAM are disabled on both CPUs until the flash operation completes.

IRAM-Safe Interrupt Handlers If you have an interrupt handler that you want to execute while a flash operation is in progress (for example, for low latency operations), set the `ESP_INTR_FLAG_IRAM` flag when the *interrupt handler is registered*.

You must ensure that all data and functions accessed by these interrupt handlers, including the ones that handlers call, are located in IRAM or DRAM.

Use the `IRAM_ATTR` attribute for functions:

```
#include "esp_attr.h"

void IRAM_ATTR gpio_isr_handler(void* arg)
{
    // ...
}
```

Use the `DRAM_ATTR` and `DRAM_STR` attributes for constant data:

```
void IRAM_ATTR gpio_isr_handler(void* arg)
{
    const static DRAM_ATTR uint8_t INDEX_DATA[] = { 45, 33, 12, 0 };
    const static char *MSG = DRAM_STR("I am a string stored in RAM");
}
```

Note that knowing which data should be marked with `DRAM_ATTR` can be hard, the compiler will sometimes recognize that a variable or expression is constant (even if it is not marked `const`) and optimize it into flash, unless it is marked with `DRAM_ATTR`.

If a function or symbol is not correctly put into IRAM/DRAM, and the interrupt handler reads from the flash cache during a flash operation, it will cause a crash due to Illegal Instruction exception (for code which should be in IRAM) or garbage data to be read (for constant data which should be in DRAM).

Partition table API

ESP-IDF projects use a partition table to maintain information about various regions of SPI flash memory (bootloader, various application binaries, data, filesystems). More information on partition tables can be found [here](#).

This component provides API functions to enumerate partitions found in the partition table and perform operations on them. These functions are declared in `esp_partition.h`:

- `esp_partition_find()` checks a partition table for entries with specific type, returns an opaque iterator.
- `esp_partition_get()` returns a structure describing the partition for a given iterator.
- `esp_partition_next()` shifts the iterator to the next found partition.
- `esp_partition_iterator_release()` releases iterator returned by `esp_partition_find`.
- `esp_partition_find_first()` - a convenience function which returns the structure describing the first partition found by `esp_partition_find`.
- `esp_partition_read()`, `esp_partition_write()`, `esp_partition_erase_range()` are equivalent to `spi_flash_read()`, `spi_flash_write()`, `spi_flash_erase_range()`, but operate within partition boundaries.

Note: Application code should mostly use these `esp_partition_*` API functions instead of lower level `spi_flash_*` API functions. Partition table API functions do bounds checking and calculate correct offsets in flash, based on data stored in a partition table.

SPI Flash Encryption

It is possible to encrypt the contents of SPI flash and have it transparently decrypted by hardware.

Refer to the [Flash Encryption documentation](#) for more details.

Memory mapping API

ESP32 features memory hardware which allows regions of flash memory to be mapped into instruction and data address spaces. This mapping works only for read operations. It is not possible to modify contents of flash memory by writing to a mapped memory region.

Mapping happens in 64KB pages. Memory mapping hardware can map up to four megabytes of flash into data address space and up to 16 megabytes of flash into instruction address space. See the technical reference manual for more details about memory mapping hardware.

Note that some 64KB pages are used to map the application itself into memory, so the actual number of available 64KB pages may be less.

Reading data from flash using a memory mapped region is the only way to decrypt contents of flash when *flash encryption* is enabled. Decryption is performed at the hardware level.

Memory mapping API are declared in `esp_spi_flash.h` and `esp_partition.h`:

- `spi_flash_mmap()` maps a region of physical flash addresses into instruction space or data space of the CPU.
- `spi_flash_munmap()` unmaps previously mapped region.
- `esp_partition_mmap()` maps part of a partition into the instruction space or data space of the CPU.

Differences between `spi_flash_mmap()` and `esp_partition_mmap()` are as follows:

- `spi_flash_mmap()` must be given a 64KB aligned physical address.
- `esp_partition_mmap()` may be given any arbitrary offset within the partition, it will adjust the returned pointer to mapped memory as necessary

Note that since memory mapping happens in 64KB blocks, it may be possible to read data outside of the partition provided to `esp_partition_mmap`.

Implementation

The `esp_flash_t` structure holds chip data as well as three important parts of this API:

1. The host driver, which provides the hardware support to access the chip;
2. The chip driver, which provides compatibility service to different chips;
3. The OS functions, provides support of some OS functions (e.g. lock, delay) in different stages (1st/2st boot, or the app).

Host driver The host driver relies on an interface (`spi_flash_host_driver_t`) defined in the `spi_flash_host_drv.h` (in the `soc/include/hal` folder). This interface provides some common functions to communicate with the chip.

In other files of the SPI HAL, some of these functions are implemented with existing ESP32 memory-spi functionalities. However due to the speed limitations of ESP32, the HAL layer can't provide high-speed implementations to some reading commands (So we didn't do it at all). The files (`memspi_host_driver.h` and `.c`) implement the high-speed version of these commands with the `common_command` function provided in the HAL, and wrap these functions as `spi_flash_host_driver_t` for upper layer to use.

You can also implement your own host driver, even with the GPIO. As long as all the functions in the `spi_flash_host_driver_t` are implemented, the `esp_flash` API can access to the flash regardless of the low-level hardware.

Chip driver The chip driver, defined in `spi_flash_chip_driver.h`, wraps basic functions provided by the host driver for the API layer to use.

Some operations need some commands to be sent first, or read some status after. Some chips need different command or value, or need special communication ways.

There is a type of chip called `generic_chip` which stands for common chips. Other special chip drivers can be developed on the base of the generic chip.

The chip driver relies on the host driver.

OS functions Currently the OS function layer provides a lock and a delay entries.

The lock is used to resolve the conflicts between the SPI chip access and other functions. E.g. the cache (used for the code and PSRAM data fetch) should be disabled when the flash chip on the SPI0/1 is being accessed. Also, some devices which don't have CS wire, or the wire is controlled by the software (e.g. SD card via SPI interface), requires the bus to be monopolized during a period.

The delay is used by some long operations which requires the master to wait or polling periodically.

The top API wraps these the chip driver and OS functions into an entire component, and also provides some argument checking.

See also

- [Partition Table documentation](#)
- [Over The Air Update \(OTA\) API](#) provides high-level API for updating app firmware stored in flash.
- [Non-Volatile Storage \(NVS\) API](#) provides a structured API for storing small pieces of data in SPI flash.

Implementation details

In order to perform some flash operations, it is necessary to make sure that both CPUs are not running any code from flash for the duration of the flash operation: - In a single-core setup, the SDK does it by disabling interrupts/scheduler before performing the flash operation. - In a dual-core setup, this is slightly more complicated as the SDK needs to make sure that the other CPU is not running any code from flash.

When SPI flash API is called on CPU A (can be PRO or APP), start the `spi_flash_op_block_func` function on CPU B using the `esp_ipc_call` API. This API wakes up a high priority task on CPU B and tells it to execute a given function, in this case, `spi_flash_op_block_func`. This function disables cache on CPU B and signals that the cache is disabled by setting the `s_flash_op_can_start` flag. Then the task on CPU A disables cache as well and proceeds to execute flash operation.

While a flash operation is running, interrupts can still run on CPUs A and B. It is assumed that all interrupt code is placed into RAM. Once the interrupt allocation API is added, a flag should be added to request the interrupt to be disabled for the duration of a flash operations.

Once the flash operation is complete, the function on CPU A sets another flag, `s_flash_op_complete`, to let the task on CPU B know that it can re-enable cache and release the CPU. Then the function on CPU A re-enables the cache on CPU A as well and returns control to the calling code.

Additionally, all API functions are protected with a mutex (`s_flash_op_mutex`).

In a single core environment (`CONFIG_FREERTOS_UNICORE` enabled), you need to disable both caches, so that no inter-CPU communication can take place.

API Reference - SPI Flash

Header File

- [spi_flash/include/esp_flash_spi_init.h](#)

Functions

`esp_err_t spi_bus_add_flash_device` (`esp_flash_t` `**out_chip`, `const`
`esp_flash_spi_device_config_t *config`)

Add a SPI Flash device onto the SPI bus.

The bus should be already initialized by `spi_bus_initialization`.

Return

- `ESP_ERR_INVALID_ARG`: `out_chip` is NULL, or some field in the config is invalid.
- `ESP_ERR_NO_MEM`: failed to allocate memory for the chip structures.
- `ESP_OK`: success.

Parameters

- `out_chip`: Pointer to hold the initialized chip.
- `config`: Configuration of the chips to initialize.

`esp_err_t spi_bus_remove_flash_device` (`esp_flash_t *chip`)

Remove a SPI Flash device from the SPI bus.

Return

- `ESP_ERR_INVALID_ARG`: The chip is invalid.
- `ESP_OK`: success.

Parameters

- `chip`: The flash device to remove.

Structures

`struct esp_flash_spi_device_config_t`

Configurations for the SPI Flash to init.

Public Members

`spi_host_device_t host_id`

Bus to use.

int `cs_id`

CS pin (signal) to use.

int `cs_io_num`

GPIO pin to output the CS signal.

`esp_flash_io_mode_t io_mode`

IO mode to read from the Flash.

`esp_flash_speed_t speed`

Speed of the Flash clock.

int `input_delay_ns`

Input delay of the data pins, in ns. Set to 0 if unknown.

Header File

- `spi_flash/include/esp_flash.h`

Functions

`esp_err_t esp_flash_init` (`esp_flash_t *chip`)

Initialise SPI flash chip interface.

This function must be called before any other API functions are called for this chip.

Note Only the `host` and `read_mode` fields of the chip structure must be initialised before this function is called. Other fields may be auto-detected if left set to zero or NULL.

Note If the `chip->drv` pointer is NULL, `chip chip_drv` will be auto-detected based on its manufacturer & product IDs. See `esp_flash_registered_flash_drivers` pointer for details of this process.

Return ESP_OK on success, or a flash error code if initialisation fails.

Parameters

- `chip`: Pointer to SPI flash chip to use. If NULL, `esp_flash_default_chip` is substituted.

bool **esp_flash_chip_driver_initialized** (const *esp_flash_t* **chip*)

Check if appropriate chip driver is set.

Return true if set, otherwise false.

Parameters

- `chip`: Pointer to SPI flash chip to use. If NULL, `esp_flash_default_chip` is substituted.

esp_err_t **esp_flash_read_id** (*esp_flash_t* **chip*, uint32_t **out_id*)

Read flash ID via the common “RDID” SPI flash command.

ID is a 24-bit value. Lower 16 bits of ‘id’ are the chip ID, upper 8 bits are the manufacturer ID.

Parameters

- `chip`: Pointer to identify flash chip. Must have been successfully initialised via `esp_flash_init()`
- [out] `out_id`: Pointer to receive ID value.

Return ESP_OK on success, or a flash error code if operation failed.

esp_err_t **esp_flash_get_size** (*esp_flash_t* **chip*, uint32_t **out_size*)

Detect flash size based on flash ID.

Note Most flash chips use a common format for flash ID, where the lower 4 bits specify the size as a power of 2. If the manufacturer doesn’t follow this convention, the size may be incorrectly detected.

Return ESP_OK on success, or a flash error code if operation failed.

Parameters

- `chip`: Pointer to identify flash chip. Must have been successfully initialised via `esp_flash_init()`
- [out] `out_size`: Detected size in bytes.

esp_err_t **esp_flash_erase_chip** (*esp_flash_t* **chip*)

Erase flash chip contents.

Return ESP_OK on success, or a flash error code if operation failed.

Parameters

- `chip`: Pointer to identify flash chip. Must have been successfully initialised via `esp_flash_init()`

esp_err_t **esp_flash_erase_region** (*esp_flash_t* **chip*, uint32_t *start*, uint32_t *len*)

Erase a region of the flash chip.

Sector size is specified in `chip->drv->sector_size` field (typically 4096 bytes.) ESP_ERR_INVALID_ARG will be returned if the start & length are not a multiple of this size.

Parameters

- `chip`: Pointer to identify flash chip. Must have been successfully initialised via `esp_flash_init()`
- `start`: Address to start erasing flash. Must be sector aligned.
- `len`: Length of region to erase. Must also be sector aligned.

Erase is performed using block (multi-sector) erases where possible (block size is specified in `chip->drv->block_erase_size` field, typically 65536 bytes). Remaining sectors are erased using individual sector erase commands.

Return ESP_OK on success, or a flash error code if operation failed.

esp_err_t **esp_flash_get_chip_write_protect** (*esp_flash_t* **chip*, bool **write_protected*)

Read if the entire chip is write protected.

Note A correct result for this flag depends on the SPI flash chip model and `chip_drv` in use (via the ‘`chip->drv`’ field).

Return ESP_OK on success, or a flash error code if operation failed.

Parameters

- `chip`: Pointer to identify flash chip. Must have been successfully initialised via `esp_flash_init()`
- [out] `write_protected`: Pointer to boolean, set to the value of the write protect flag.

esp_err_t **esp_flash_set_chip_write_protect** (*esp_flash_t* **chip*, bool *write_protect*)

Set write protection for the SPI flash chip.

Some SPI flash chips may require a power cycle before write protect status can be cleared. Otherwise, write protection can be removed via a follow-up call to this function.

Note Correct behaviour of this function depends on the SPI flash chip model and `chip_drv` in use (via the `'chip->drv'` field).

Parameters

- `chip`: Pointer to identify flash chip. Must have been successfully initialised via `esp_flash_init()`
- `write_protect`: Boolean value for the write protect flag

Return `ESP_OK` on success, or a flash error code if operation failed.

```
esp_err_t esp_flash_get_protectable_regions(const esp_flash_t *chip, const
                                           esp_flash_region_t **out_regions, uint32_t
                                           *out_num_regions)
```

Read the list of individually protectable regions of this SPI flash chip.

Note Correct behaviour of this function depends on the SPI flash chip model and `chip_drv` in use (via the `'chip->drv'` field).

Return `ESP_OK` on success, or a flash error code if operation failed.

Parameters

- `chip`: Pointer to identify flash chip. Must have been successfully initialised via `esp_flash_init()`
- `[out] out_regions`: Pointer to receive a pointer to the array of protectable regions of the chip.
- `[out] out_num_regions`: Pointer to an integer receiving the count of protectable regions in the array returned in `'regions'` .

```
esp_err_t esp_flash_get_protected_region(esp_flash_t *chip, const esp_flash_region_t
                                          *region, bool *out_protected)
```

Detect if a region of the SPI flash chip is protected.

Note It is possible for this result to be false and write operations to still fail, if protection is enabled for the entire chip.

Note Correct behaviour of this function depends on the SPI flash chip model and `chip_drv` in use (via the `'chip->drv'` field).

Return `ESP_OK` on success, or a flash error code if operation failed.

Parameters

- `chip`: Pointer to identify flash chip. Must have been successfully initialised via `esp_flash_init()`
- `region`: Pointer to a struct describing a protected region. This must match one of the regions returned from `esp_flash_get_protectable_regions(...)`.
- `[out] out_protected`: Pointer to a flag which is set based on the protected status for this region.

```
esp_err_t esp_flash_set_protected_region(esp_flash_t *chip, const esp_flash_region_t
                                          *region, bool protect)
```

Update the protected status for a region of the SPI flash chip.

Note It is possible for the region protection flag to be cleared and write operations to still fail, if protection is enabled for the entire chip.

Note Correct behaviour of this function depends on the SPI flash chip model and `chip_drv` in use (via the `'chip->drv'` field).

Return `ESP_OK` on success, or a flash error code if operation failed.

Parameters

- `chip`: Pointer to identify flash chip. Must have been successfully initialised via `esp_flash_init()`
- `region`: Pointer to a struct describing a protected region. This must match one of the regions returned from `esp_flash_get_protectable_regions(...)`.
- `protect`: Write protection flag to set.

```
esp_err_t esp_flash_read(esp_flash_t *chip, void *buffer, uint32_t address, uint32_t length)
```

Read data from the SPI flash chip.

There are no alignment constraints on `buffer`, `address` or `length`.

Parameters

- `chip`: Pointer to identify flash chip. Must have been successfully initialised via `esp_flash_init()`

- `buffer`: Pointer to a buffer where the data will be read. To get better performance, this should be in the DRAM and word aligned.
- `address`: Address on flash to read from. Must be less than `chip->size` field.
- `length`: Length (in bytes) of data to read.

Note If on-chip flash encryption is used, this function returns raw (ie encrypted) data. Use the flash cache to transparently decrypt data.

Return

- `ESP_OK`: success
- `ESP_ERR_NO_MEM`: Buffer is in external PSRAM which cannot be concurrently accessed, and a temporary internal buffer could not be allocated.
- or a flash error code if operation failed.

`esp_err_t esp_flash_write(esp_flash_t *chip, const void *buffer, uint32_t address, uint32_t length)`
Write data to the SPI flash chip.

There are no alignment constraints on `buffer`, `address` or `length`.

Parameters

- `chip`: Pointer to identify flash chip. Must have been successfully initialised via `esp_flash_init()`
- `address`: Address on flash to write to. Must be previously erased (SPI NOR flash can only write bits 1->0).
- `buffer`: Pointer to a buffer with the data to write. To get better performance, this should be in the DRAM and word aligned.
- `length`: Length (in bytes) of data to write.

Return `ESP_OK` on success, or a flash error code if operation failed.

`esp_err_t esp_flash_write_encrypted(esp_flash_t *chip, uint32_t address, const void *buffer, uint32_t length)`

Encrypted and write data to the SPI flash chip using on-chip hardware flash encryption.

Note Both `address` & `length` must be 16 byte aligned, as this is the encryption block size

Return

- `ESP_OK`: on success
- `ESP_ERR_NOT_SUPPORTED`: encrypted write not supported for this chip.
- `ESP_ERR_INVALID_ARG`: Either the `address`, `buffer` or `length` is invalid.
- or other flash error code from `spi_flash_write_encrypted()`.

Parameters

- `chip`: Pointer to identify flash chip. Must be `NULL` (the main flash chip). For other chips, encrypted write is not supported.
- `address`: Address on flash to write to. 16 byte aligned. Must be previously erased (SPI NOR flash can only write bits 1->0).
- `buffer`: Pointer to a buffer with the data to write.
- `length`: Length (in bytes) of data to write. 16 byte aligned.

`esp_err_t esp_flash_read_encrypted(esp_flash_t *chip, uint32_t address, void *out_buffer, uint32_t length)`

Read and decrypt data from the SPI flash chip using on-chip hardware flash encryption.

Return

- `ESP_OK`: on success
- `ESP_ERR_NOT_SUPPORTED`: encrypted read not supported for this chip.
- or other flash error code from `spi_flash_read_encrypted()`.

Parameters

- `chip`: Pointer to identify flash chip. Must be `NULL` (the main flash chip). For other chips, encrypted read is not supported.
- `address`: Address on flash to read from.
- `out_buffer`: Pointer to a buffer for the data to read to.
- `length`: Length (in bytes) of data to read.

`static bool esp_flash_is_quad_mode(const esp_flash_t *chip)`

Returns true if chip is configured for Quad I/O or Quad Fast Read.

Return true if flash works in quad mode, otherwise false

Parameters

- `chip`: Pointer to SPI flash chip to use. If NULL, `esp_flash_default_chip` is substituted.

Structures

struct esp_flash_region_t

Structure for describing a region of flash.

Public Members

`uint32_t offset`

Start address of this region.

`uint32_t size`

Size of the region.

struct esp_flash_os_functions_t

OS-level integration hooks for accessing flash chips inside a running OS

Public Members

`esp_err_t (*start)(void *arg)`

Called before commencing any flash operation. Does not need to be recursive (ie is called at most once for each call to ‘end’).

`esp_err_t (*end)(void *arg)`

Called after completing any flash operation.

`esp_err_t (*region_protected)(void *arg, size_t start_addr, size_t size)`

Called before any erase/write operations to check whether the region is limited by the OS

`esp_err_t (*delay_us)(void *arg, unsigned us)`

Delay for at least ‘us’ microseconds. Called in between ‘start’ and ‘end’.

`esp_err_t (*yield)(void *arg)`

Yield to other tasks. Called during erase operations.

struct esp_flash_t

Structure to describe a SPI flash chip connected to the system.

Structure must be initialized before use (passed to `esp_flash_init()`).

Public Members

`spi_flash_host_driver_t *host`

Pointer to hardware-specific “host_driver” structure. Must be initialized before used.

`const spi_flash_chip_t *chip_drv`

Pointer to chip-model-specific “adapter” structure. If NULL, will be detected during initialisation.

`const esp_flash_os_functions_t *os_func`

Pointer to os-specific hook structure. Call `esp_flash_init_os_functions()` to setup this field, after the host is properly initialized.

`void *os_func_data`

Pointer to argument for os-specific hooks. Left NULL and will be initialized with `os_func`.

`esp_flash_io_mode_t read_mode`

Configured SPI flash read mode. Set before `esp_flash_init` is called.

`uint32_t size`

Size of SPI flash in bytes. If 0, size will be detected during initialisation.

uint32_t **chip_id**
Detected chip id.

Type Definitions

```
typedef struct spi_flash_chip_t spi_flash_chip_t
typedef struct esp_flash_t esp_flash_t
```

Header File

- [soc/include/hal/spi_flash_types.h](#)

Structures

struct spi_flash_trans_t
Definition of a common transaction. Also holds the return value.

Public Members

uint8_t **command**
Command to send, always 8bits.

uint8_t **mosi_len**
Output data length, in bytes.

uint8_t **miso_len**
Input data length, in bytes.

uint8_t **address_bitlen**
Length of address in bits, set to 0 if command does not need an address.

uint32_t **address**
Address to perform operation on.

const uint8_t ***mosi_data**
Output data to salve.

uint8_t ***miso_data**
[out] Input data from slave, little endian

struct spi_flash_host_driver_t
Host driver configuration and context structure.

Public Members

void ***driver_data**
Configuration and static data used by the specific host driver. The type is determined by the host driver.

esp_err_t (***dev_config**) (*spi_flash_host_driver_t* *driver)
Configure the device-related register before transactions. This saves some time to re-configure those registers when we send continuously

esp_err_t (***common_command**) (*spi_flash_host_driver_t* *driver, *spi_flash_trans_t* *t)
Send an user-defined spi transaction to the device.

esp_err_t (***read_id**) (*spi_flash_host_driver_t* *driver, uint32_t *id)
Read flash ID.

void (***erase_chip**) (*spi_flash_host_driver_t* *driver)
Erase whole flash chip.

void (***erase_sector**) (*spi_flash_host_driver_t* *driver, uint32_t start_address)
Erase a specific sector by its start address.

`void (*erase_block) (spi_flash_host_driver_t *driver, uint32_t start_address)`
Erase a specific block by its start address.

`esp_err_t (*read_status) (spi_flash_host_driver_t *driver, uint8_t *out_sr)`
Read the status of the flash chip.

`esp_err_t (*set_write_protect) (spi_flash_host_driver_t *driver, bool wp)`
Disable write protection.

`void (*program_page) (spi_flash_host_driver_t *driver, const void *buffer, uint32_t address, uint32_t length)`
Program a page of the flash. Check `max_write_bytes` for the maximum allowed writing length.

`bool (*supports_direct_write) (spi_flash_host_driver_t *driver, const void *p)`
Check whether need to allocate new buffer to write

`bool (*supports_direct_read) (spi_flash_host_driver_t *driver, const void *p)`
Check whether need to allocate new buffer to read

`int max_write_bytes`
maximum length of program_page

`esp_err_t (*read) (spi_flash_host_driver_t *driver, void *buffer, uint32_t address, uint32_t read_len)`
Read data from the flash. Check `max_read_bytes` for the maximum allowed reading length.

`int max_read_bytes`
maximum length of read

`bool (*host_idle) (spi_flash_host_driver_t *driver)`
Check whether the host is idle to perform new operations.

`esp_err_t (*configure_host_io_mode) (spi_flash_host_driver_t *driver, uint32_t command, uint32_t addr_bitlen, int dummy_bitlen_base, esp_flash_io_mode_t io_mode)`
Configure the host to work at different read mode. Responsible to compensate the timing and set IO mode.

`void (*poll_cmd_done) (spi_flash_host_driver_t *driver)`
Internal use, poll the HW until the last operation is done.

`esp_err_t (*flush_cache) (spi_flash_host_driver_t *driver, uint32_t addr, uint32_t size)`
For some host (SPI1), they are shared with a cache. When the data is modified, the cache needs to be flushed. Left NULL if not supported.

Macros

ESP_FLASH_SPEED_MIN

Lowest speed supported by the driver, currently 5 MHz.

SPI_FLASH_READ_MODE_MIN

Slowest io mode supported by ESP32, currently SlowRd.

Type Definitions

```
typedef struct spi_flash_host_driver_t spi_flash_host_driver_t
```

Enumerations

enum esp_flash_speed_t

SPI flash clock speed values, always refer to them by the enum rather than the actual value (more speed may be appended into the list).

A strategy to select the maximum allowed speed is to enumerate from the `ESP_FLASH_SPEED_MAX-1` or highest frequency supported by your flash, and decrease the speed until the probing success.

Values:

ESP_FLASH_5MHZ = 0

The flash runs under 5MHz.

ESP_FLASH_10MHZ

The flash runs under 10MHz.

ESP_FLASH_20MHZ

The flash runs under 20MHz.

ESP_FLASH_26MHZ

The flash runs under 26MHz.

ESP_FLASH_40MHZ

The flash runs under 40MHz.

ESP_FLASH_80MHZ

The flash runs under 80MHz.

ESP_FLASH_SPEED_MAX

The maximum frequency supported by the host is ESP_FLASH_SPEED_MAX-1.

enum esp_flash_io_mode_t

Mode used for reading from SPI flash.

Values:

SPI_FLASH_SLOWRD = 0

Data read using single I/O, some limits on speed.

SPI_FLASH_FASTRD

Data read using single I/O, no limit on speed.

SPI_FLASH_DOUT

Data read using dual I/O.

SPI_FLASH_DIO

Both address & data transferred using dual I/O.

SPI_FLASH_QOUT

Data read using quad I/O.

SPI_FLASH_QIO

Both address & data transferred using quad I/O.

SPI_FLASH_READ_MODE_MAX

The fastest io mode supported by the host is ESP_FLASH_READ_MODE_MAX-1.

API Reference - Partition Table

Header File

- `spi_flash/include/esp_partition.h`

Functions

`esp_partition_iterator_t esp_partition_find(esp_partition_type_t type, esp_partition_subtype_t subtype, const char *label)`

Find partition based on one or more parameters.

Return iterator which can be used to enumerate all the partitions found, or NULL if no partitions were found.

Iterator obtained through this function has to be released using `esp_partition_iterator_release` when not used any more.

Parameters

- `type`: Partition type, one of `esp_partition_type_t` values
- `subtype`: Partition subtype, one of `esp_partition_subtype_t` values. To find all partitions of given type, use `ESP_PARTITION_SUBTYPE_ANY`.

- `label`: (optional) Partition label. Set this value if looking for partition with a specific name. Pass NULL otherwise.

```
const esp_partition_t *esp_partition_find_first (esp_partition_type_t type,
                                             esp_partition_subtype_t subtype, const
                                             char *label)
```

Find first partition based on one or more parameters.

Return pointer to *esp_partition_t* structure, or NULL if no partition is found. This pointer is valid for the lifetime of the application.

Parameters

- `type`: Partition type, one of *esp_partition_type_t* values
- `subtype`: Partition subtype, one of *esp_partition_subtype_t* values. To find all partitions of given type, use `ESP_PARTITION_SUBTYPE_ANY`.
- `label`: (optional) Partition label. Set this value if looking for partition with a specific name. Pass NULL otherwise.

```
const esp_partition_t *esp_partition_get (esp_partition_iterator_t iterator)
```

Get *esp_partition_t* structure for given partition.

Return pointer to *esp_partition_t* structure. This pointer is valid for the lifetime of the application.

Parameters

- `iterator`: Iterator obtained using `esp_partition_find`. Must be non-NULL.

```
esp_partition_iterator_t esp_partition_next (esp_partition_iterator_t iterator)
```

Move partition iterator to the next partition found.

Any copies of the iterator will be invalid after this call.

Return NULL if no partition was found, valid *esp_partition_iterator_t* otherwise.

Parameters

- `iterator`: Iterator obtained using `esp_partition_find`. Must be non-NULL.

```
void esp_partition_iterator_release (esp_partition_iterator_t iterator)
```

Release partition iterator.

Parameters

- `iterator`: Iterator obtained using `esp_partition_find`. Must be non-NULL.

```
const esp_partition_t *esp_partition_verify (const esp_partition_t *partition)
```

Verify partition data.

Given a pointer to partition data, verify this partition exists in the partition table (all fields match.)

This function is also useful to take partition data which may be in a RAM buffer and convert it to a pointer to the permanent partition data stored in flash.

Pointers returned from this function can be compared directly to the address of any pointer returned from `esp_partition_get()`, as a test for equality.

Return

- If partition not found, returns NULL.
- If found, returns a pointer to the *esp_partition_t* structure in flash. This pointer is always valid for the lifetime of the application.

Parameters

- `partition`: Pointer to partition data to verify. Must be non-NULL. All fields of this structure must match the partition table entry in flash for this function to return a successful match.

```
esp_err_t esp_partition_read (const esp_partition_t *partition, size_t src_offset, void *dst, size_t
                               size)
```

Read data from the partition.

Return `ESP_OK`, if data was read successfully; `ESP_ERR_INVALID_ARG`, if `src_offset` exceeds partition size; `ESP_ERR_INVALID_SIZE`, if read would go out of bounds of the partition; or one of error codes from lower-level flash driver.

Parameters

- `partition`: Pointer to partition structure obtained using `esp_partition_find_first` or `esp_partition_get`. Must be non-NULL.
- `dst`: Pointer to the buffer where data should be stored. Pointer must be non-NULL and buffer must be at least ‘size’ bytes long.
- `src_offset`: Address of the data to be read, relative to the beginning of the partition.
- `size`: Size of data to be read, in bytes.

esp_err_t **esp_partition_write** (**const** *esp_partition_t* **partition*, *size_t* *dst_offset*, **const** void **src*, *size_t* *size*)

Write data to the partition.

Before writing data to flash, corresponding region of flash needs to be erased. This can be done using `esp_partition_erase_range` function.

Partitions marked with an encryption flag will automatically be written via the `spi_flash_write_encrypted()` function. If writing to an encrypted partition, all write offsets and lengths must be multiples of 16 bytes. See the `spi_flash_write_encrypted()` function for more details. Unencrypted partitions do not have this restriction.

Note Prior to writing to flash memory, make sure it has been erased with `esp_partition_erase_range` call.

Return `ESP_OK`, if data was written successfully; `ESP_ERR_INVALID_ARG`, if `dst_offset` exceeds partition size; `ESP_ERR_INVALID_SIZE`, if write would go out of bounds of the partition; or one of error codes from lower-level flash driver.

Parameters

- `partition`: Pointer to partition structure obtained using `esp_partition_find_first` or `esp_partition_get`. Must be non-NULL.
- `dst_offset`: Address where the data should be written, relative to the beginning of the partition.
- `src`: Pointer to the source buffer. Pointer must be non-NULL and buffer must be at least ‘size’ bytes long.
- `size`: Size of data to be written, in bytes.

esp_err_t **esp_partition_erase_range** (**const** *esp_partition_t* **partition*, *size_t* *offset*, *size_t* *size*)

Erase part of the partition.

Return `ESP_OK`, if the range was erased successfully; `ESP_ERR_INVALID_ARG`, if iterator or `dst` are NULL; `ESP_ERR_INVALID_SIZE`, if erase would go out of bounds of the partition; or one of error codes from lower-level flash driver.

Parameters

- `partition`: Pointer to partition structure obtained using `esp_partition_find_first` or `esp_partition_get`. Must be non-NULL.
- `offset`: Offset from the beginning of partition where erase operation should start. Must be aligned to 4 kilobytes.
- `size`: Size of the range which should be erased, in bytes. Must be divisible by 4 kilobytes.

esp_err_t **esp_partition_mmap** (**const** *esp_partition_t* **partition*, *size_t* *offset*, *size_t* *size*, *spi_flash_mmap_memory_t* *memory*, **const** void ***out_ptr*, *spi_flash_mmap_handle_t* **out_handle*)

Configure MMU to map partition into data memory.

Unlike `spi_flash_mmap` function, which requires a 64kB aligned base address, this function doesn't impose such a requirement. If offset results in a flash address which is not aligned to 64kB boundary, address will be rounded to the lower 64kB boundary, so that mapped region includes requested range. Pointer returned via `out_ptr` argument will be adjusted to point to the requested offset (not necessarily to the beginning of mmap-ed region).

To release mapped memory, pass handle returned via `out_handle` argument to `spi_flash_munmap` function.

Return `ESP_OK`, if successful

Parameters

- `partition`: Pointer to partition structure obtained using `esp_partition_find_first` or `esp_partition_get`. Must be non-NULL.
- `offset`: Offset from the beginning of partition where mapping should start.
- `size`: Size of the area to be mapped.
- `memory`: Memory space where the region should be mapped

- `out_ptr`: Output, pointer to the mapped memory region
- `out_handle`: Output, handle which should be used for `spi_flash_munmap` call

`esp_err_t esp_partition_get_sha256 (const esp_partition_t *partition, uint8_t *sha_256)`

Get SHA-256 digest for required partition.

For apps with SHA-256 appended to the app image, the result is the appended SHA-256 value for the app image content. The hash is verified before returning, if app content is invalid then the function returns `ESP_ERR_IMAGE_INVALID`. For apps without SHA-256 appended to the image, the result is the SHA-256 of all bytes in the app image. For other partition types, the result is the SHA-256 of the entire partition.

Return

- `ESP_OK`: In case of successful operation.
- `ESP_ERR_INVALID_ARG`: The size was 0 or the `sha_256` was NULL.
- `ESP_ERR_NO_MEM`: Cannot allocate memory for sha256 operation.
- `ESP_ERR_IMAGE_INVALID`: App partition doesn't contain a valid app image.
- `ESP_FAIL`: An allocation error occurred.

Parameters

- `[in] partition`: Pointer to info for partition containing app or data. (fields: address, size and type, are required to be filled).
- `[out] sha_256`: Returned SHA-256 digest for a given partition.

`bool esp_partition_check_identity (const esp_partition_t *partition_1, const esp_partition_t *partition_2)`

Check for the identity of two partitions by SHA-256 digest.

Return

- `True`: In case of the two firmware is equal.
- `False`: Otherwise

Parameters

- `[in] partition_1`: Pointer to info for partition 1 containing app or data. (fields: address, size and type, are required to be filled).
- `[in] partition_2`: Pointer to info for partition 2 containing app or data. (fields: address, size and type, are required to be filled).

`esp_err_t esp_partition_register_external (esp_flash_t *flash_chip, size_t offset, size_t size, const char *label, esp_partition_type_t type, esp_partition_subtype_t subtype, const esp_partition_t **out_partition)`

Register a partition on an external flash chip.

This API allows designating certain areas of external flash chips (identified by the `esp_flash_t` structure) as partitions. This allows using them with components which access SPI flash through the `esp_partition` API.

Return

- `ESP_OK` on success
- `ESP_ERR_NOT_SUPPORTED` if `CONFIG_CONFIG_SPI_FLASH_USE_LEGACY_IMPL` is enabled
- `ESP_ERR_NO_MEM` if memory allocation has failed
- `ESP_ERR_INVALID_ARG` if the new partition overlaps another partition on the same flash chip
- `ESP_ERR_INVALID_SIZE` if the partition doesn't fit into the flash chip size

Parameters

- `flash_chip`: Pointer to the structure identifying the flash chip
- `offset`: Address in bytes, where the partition starts
- `size`: Size of the partition in bytes
- `label`: Partition name
- `type`: One of the partition types (`ESP_PARTITION_TYPE_*`). Note that applications can not be booted from external flash chips, so using `ESP_PARTITION_TYPE_APP` is not supported.
- `subtype`: One of the partition subtypes (`ESP_PARTITION_SUBTYPE_*`)
- `[out] out_partition`: Output, if non-NULL, receives the pointer to the resulting `esp_partition_t` structure

esp_err_t **esp_partition_deregister_external** (const *esp_partition_t* *partition)

Deregister the partition previously registered using `esp_partition_register_external`.

Return

- ESP_OK on success
- ESP_ERR_NOT_FOUND if the partition pointer is not found
- ESP_ERR_INVALID_ARG if the partition comes from the partition table
- ESP_ERR_INVALID_ARG if the partition was not registered using `esp_partition_register_external` function.

Parameters

- `partition`: pointer to the partition structure obtained from `esp_partition_register_external`,

Structures

struct esp_partition_t

partition information structure

This is not the format in flash, that format is `esp_partition_info_t`.

However, this is the format used by this API.

Public Members

esp_flash_t ***flash_chip**

SPI flash chip on which the partition resides

esp_partition_type_t **type**

partition type (app/data)

esp_partition_subtype_t **subtype**

partition subtype

uint32_t **address**

starting address of the partition in flash

uint32_t **size**

size of the partition, in bytes

char **label**[17]

partition label, zero-terminated ASCII string

bool **encrypted**

flag is set to true if partition is encrypted

Macros

ESP_PARTITION_SUBTYPE_OTA (i)

Convenience macro to get `esp_partition_subtype_t` value for the i-th OTA partition.

Type Definitions

typedef struct esp_partition_iterator_opaque_ ***esp_partition_iterator_t**

Opaque partition iterator type.

Enumerations

enum esp_partition_type_t

Partition type.

Note Keep this enum in sync with PartitionDefinition class `gen_esp32part.py`

Values:

ESP_PARTITION_TYPE_APP = 0x00

Application partition type.

ESP_PARTITION_TYPE_DATA = 0x01

Data partition type.

enum esp_partition_subtype_t

Partition subtype.

Note Keep this enum in sync with PartitionDefinition class gen_esp32part.py

Values:

ESP_PARTITION_SUBTYPE_APP_FACTORY = 0x00

Factory application partition.

ESP_PARTITION_SUBTYPE_APP_OTA_MIN = 0x10

Base for OTA partition subtypes.

ESP_PARTITION_SUBTYPE_APP_OTA_0 = *ESP_PARTITION_SUBTYPE_APP_OTA_MIN* + 0

OTA partition 0.

ESP_PARTITION_SUBTYPE_APP_OTA_1 = *ESP_PARTITION_SUBTYPE_APP_OTA_MIN* + 1

OTA partition 1.

ESP_PARTITION_SUBTYPE_APP_OTA_2 = *ESP_PARTITION_SUBTYPE_APP_OTA_MIN* + 2

OTA partition 2.

ESP_PARTITION_SUBTYPE_APP_OTA_3 = *ESP_PARTITION_SUBTYPE_APP_OTA_MIN* + 3

OTA partition 3.

ESP_PARTITION_SUBTYPE_APP_OTA_4 = *ESP_PARTITION_SUBTYPE_APP_OTA_MIN* + 4

OTA partition 4.

ESP_PARTITION_SUBTYPE_APP_OTA_5 = *ESP_PARTITION_SUBTYPE_APP_OTA_MIN* + 5

OTA partition 5.

ESP_PARTITION_SUBTYPE_APP_OTA_6 = *ESP_PARTITION_SUBTYPE_APP_OTA_MIN* + 6

OTA partition 6.

ESP_PARTITION_SUBTYPE_APP_OTA_7 = *ESP_PARTITION_SUBTYPE_APP_OTA_MIN* + 7

OTA partition 7.

ESP_PARTITION_SUBTYPE_APP_OTA_8 = *ESP_PARTITION_SUBTYPE_APP_OTA_MIN* + 8

OTA partition 8.

ESP_PARTITION_SUBTYPE_APP_OTA_9 = *ESP_PARTITION_SUBTYPE_APP_OTA_MIN* + 9

OTA partition 9.

ESP_PARTITION_SUBTYPE_APP_OTA_10 = *ESP_PARTITION_SUBTYPE_APP_OTA_MIN* + 10

OTA partition 10.

ESP_PARTITION_SUBTYPE_APP_OTA_11 = *ESP_PARTITION_SUBTYPE_APP_OTA_MIN* + 11

OTA partition 11.

ESP_PARTITION_SUBTYPE_APP_OTA_12 = *ESP_PARTITION_SUBTYPE_APP_OTA_MIN* + 12

OTA partition 12.

ESP_PARTITION_SUBTYPE_APP_OTA_13 = *ESP_PARTITION_SUBTYPE_APP_OTA_MIN* + 13

OTA partition 13.

ESP_PARTITION_SUBTYPE_APP_OTA_14 = *ESP_PARTITION_SUBTYPE_APP_OTA_MIN* + 14

OTA partition 14.

ESP_PARTITION_SUBTYPE_APP_OTA_15 = *ESP_PARTITION_SUBTYPE_APP_OTA_MIN* + 15

OTA partition 15.

ESP_PARTITION_SUBTYPE_APP_OTA_MAX = *ESP_PARTITION_SUBTYPE_APP_OTA_MIN* + 16

Max subtype of OTA partition.

ESP_PARTITION_SUBTYPE_APP_TEST = 0x20

Test application partition.

ESP_PARTITION_SUBTYPE_DATA_OTA = 0x00
OTA selection partition.

ESP_PARTITION_SUBTYPE_DATA_PHY = 0x01
PHY init data partition.

ESP_PARTITION_SUBTYPE_DATA_NVS = 0x02
NVS partition.

ESP_PARTITION_SUBTYPE_DATA_COREDUMP = 0x03
COREDUMP partition.

ESP_PARTITION_SUBTYPE_DATA_NVS_KEYS = 0x04
Partition for NVS keys.

ESP_PARTITION_SUBTYPE_DATA_EFUSE_EM = 0x05
Partition for emulate eFuse bits.

ESP_PARTITION_SUBTYPE_DATA_UNDEFINED = 0x06
Undefined (or unspecified) data partition.

ESP_PARTITION_SUBTYPE_DATA_ESPHTTPD = 0x80
ESPHTTPD partition.

ESP_PARTITION_SUBTYPE_DATA_FAT = 0x81
FAT partition.

ESP_PARTITION_SUBTYPE_DATA_SPIFFS = 0x82
SPIFFS partition.

ESP_PARTITION_SUBTYPE_ANY = 0xff
Used to search for partitions with any subtype.

API Reference - Flash Encrypt

Header File

- [bootloader_support/include/esp_flash_encrypt.h](#)

Functions

static bool esp_flash_encryption_enabled (void)

Is flash encryption currently enabled in hardware?

Flash encryption is enabled if the FLASH_CRYPT_CNT efuse has an odd number of bits set.

Return true if flash encryption is enabled.

esp_err_t **esp_flash_encrypt_check_and_update** (void)

esp_err_t **esp_flash_encrypt_region** (uint32_t *src_addr*, size_t *data_length*)

Encrypt-in-place a block of flash sectors.

Note This function resets RTC_WDT between operations with sectors.

Return ESP_OK if all operations succeeded, ESP_ERR_FLASH_OP_FAIL if SPI flash fails, ESP_ERR_FLASH_OP_TIMEOUT if flash times out.

Parameters

- *src_addr*: Source offset in flash. Should be multiple of 4096 bytes.
- *data_length*: Length of data to encrypt in bytes. Will be rounded up to next multiple of 4096 bytes.

void **esp_flash_write_protect_crypt_cnt** (void)

Write protect FLASH_CRYPT_CNT.

Intended to be called as a part of boot process if flash encryption is enabled but secure boot is not used. This should protect against serial re-flashing of an unauthorised code in absence of secure boot.

Note On ESP32 V3 only, write protecting FLASH_CRYPT_CNT will also prevent disabling UART Download Mode. If both are wanted, call `esp_efuse_disable_rom_download_mode()` before calling this function.

`esp_flash_enc_mode_t` **esp_get_flash_encryption_mode** (void)

Return the flash encryption mode.

The API is called during boot process but can also be called by application to check the current flash encryption mode of ESP32

Return

void **esp_flash_encryption_init_checks** (void)

Check the flash encryption mode during startup.

Verifies the flash encryption config during startup:

Note This function is called automatically during app startup, it doesn't need to be called from the app.

- Correct any insecure flash encryption settings if hardware Secure Boot is enabled.
- Log warnings if the efuse config doesn't match the project config in any way

Enumerations

enum **esp_flash_enc_mode_t**

Values:

ESP_FLASH_ENC_MODE_DISABLED

ESP_FLASH_ENC_MODE_DEVELOPMENT

ESP_FLASH_ENC_MODE_RELEASE

2.6.7 SPIFFS Filesystem

Overview

SPIFFS is a file system intended for SPI NOR flash devices on embedded targets. It supports wear levelling, file system consistency checks, and more.

Notes

- Currently, SPIFFS does not support directories, it produces a flat structure. If SPIFFS is mounted under `/spiffs`, then creating a file with the path `/spiffs/tmp/myfile.txt` will create a file called `/tmp/myfile.txt` in SPIFFS, instead of `myfile.txt` in the directory `/spiffs/tmp`.
- It is not a real-time stack. One write operation might take much longer than another.
- For now, it does not detect or handle bad blocks.

Tools

spiffsgen.py `spiffsgen.py` is a write-only Python SPIFFS implementation used to create filesystem images from the contents of a host folder. To use `spiffsgen.py`, open Terminal and run:

```
python spiffsgen.py <image_size> <base_dir> <output_file>
```

The required arguments are as follows:

- **image_size**: size of the partition onto which the created SPIFFS image will be flashed.
- **base_dir**: directory for which the SPIFFS image needs to be created.
- **output_file**: SPIFFS image output file.

There are also other arguments that control image generation. Documentation on these arguments can be found in the tool's help:

```
python spiffsgen.py --help
```

These optional arguments correspond to a possible SPIFFS build configuration. To generate the right image, please make sure that you use the same arguments/configuration as were used to build SPIFFS. As a guide, the help output indicates the SPIFFS build configuration to which the argument corresponds. In cases when these arguments are not specified, the default values shown in the help output will be used.

When the image is created, it can be flashed using `esptool.py` or `parttool.py`.

Aside from invoking the `spiffsgen.py` standalone by manually running it from the command line or a script, it is also possible to invoke `spiffsgen.py` directly from the build system by calling `spiffs_create_partition_image`.

Make:

```
SPIFFS_IMAGE_FLASH_IN_PROJECT := ...
SPIFFS_IMAGE_DEPENDS := ...
$(eval $(call spiffs_create_partition_image,<partition>,<base_dir>))
```

CMake:

```
spiffs_create_partition_image(<partition> <base_dir> [FLASH_IN_PROJECT] [DEPENDS_
→dep dep dep...])
```

This is more convenient as the build configuration is automatically passed to the tool, ensuring that the generated image is valid for that build. An example of this is while the `image_size` is required for the standalone invocation, only the `partition` name is required when using `spiffs_create_partition_image`—the image size is automatically obtained from the project's partition table.

Due to the differences in structure between Make and CMake, it is important to note that: - for Make `spiffs_create_partition_image` must be called from the project Makefile - for CMake `spiffs_create_partition_image` must be called from one of the component CMakeLists.txt files

Optionally, user can opt to have the image automatically flashed together with the app binaries, partition tables, etc. on `idf.py flash` or `make flash` by specifying `FLASH_IN_PROJECT`. For example,

in Make:

```
SPIFFS_IMAGE_FLASH_IN_PROJECT := 1
$(eval $(call spiffs_create_partition_image,<partition>,<base_dir>))
```

in CMake:

```
spiffs_create_partition_image(my_spiffs_partition my_folder FLASH_IN_PROJECT)
```

If `FLASH_IN_PROJECT/SPIFFS_IMAGE_FLASH_IN_PROJECT` is not specified, the image will still be generated, but you will have to flash it manually using `esptool.py`, `parttool.py`, or a custom build system target.

There are cases where the contents of the base directory itself is generated at build time. Users can use `DEPENDS/SPIFFS_IMAGE_DEPENDS` to specify targets that should be executed before generating the image.

in Make:

```
dep:
    ...

SPIFFS_IMAGE_DEPENDS := dep
$(eval $(call spiffs_create_partition_image,<partition>,<base_dir>))
```

in CMake:

```
add_custom_target(dep COMMAND ...)  
spiffs_create_partition_image(my_spiffs_partition my_folder DEPENDS dep)
```

For an example, see [storage/spiffsgen](#).

mkspiffs Another tool for creating SPIFFS partition images is [mkspiffs](#). Similar to `spiffsgen.py`, it can be used to create an image from a given folder and then flash that image using `esptool.py`

For that, you need to obtain the following parameters:

- **Block Size:** 4096 (standard for SPI Flash)
- **Page Size:** 256 (standard for SPI Flash)
- **Image Size:** Size of the partition in bytes (can be obtained from a partition table)
- **Partition Offset:** Starting address of the partition (can be obtained from a partition table)

To pack a folder into a 1-Megabyte image, run:

```
mkspiffs -c [src_folder] -b 4096 -p 256 -s 0x100000 spiffs.bin
```

To flash the image onto ESP32 at offset 0x110000, run:

```
python esptool.py --chip esp32 --port [port] --baud [baud] write_flash -z 0x110000_  
↪spiffs.bin
```

Notes on which SPIFFS tool to use The two tools presented above offer very similar functionality. However, there are reasons to prefer one over the other, depending on the use case.

Use `spiffsgen.py` in the following cases: 1. If you want to simply generate a SPIFFS image during the build. `spiffsgen.py` makes it very convenient by providing functions/commands from the build system itself. 2. If the host has no C/C++ compiler available, because `spiffsgen.py` does not require compilation.

Use `mkspiffs` in the following cases: 1. If you need to unpack SPIFFS images in addition to image generation. For now, it is not possible with `spiffsgen.py`. 2. If you have an environment where a Python interpreter is not available, but a host compiler is available. Otherwise, a pre-compiled `mkspiffs` binary can do the job. However, there is no build system integration for `mkspiffs` and the user has to do the corresponding work: compiling `mk-spiffs` during build (if a pre-compiled binary is not used), creating build rules/targets for the output files, passing proper parameters to the tool, etc.

See also

- [Partition Table documentation](#)

Application Example

An example of using SPIFFS is provided in the [storage/spiffs](#) directory. This example initializes and mounts a SPIFFS partition, then writes and reads data from it using POSIX and C library APIs. See the `README.md` file in the example directory for more information.

High-level API Reference

Header File

- [spiffs/include/esp_spiffs.h](#)

Functions

esp_err_t **esp_vfs_spiffs_register** (**const** *esp_vfs_spiffs_conf_t* *conf)

Register and mount SPIFFS to VFS with given path prefix.

Return

- ESP_OK if success
- ESP_ERR_NO_MEM if objects could not be allocated
- ESP_ERR_INVALID_STATE if already mounted or partition is encrypted
- ESP_ERR_NOT_FOUND if partition for SPIFFS was not found
- ESP_FAIL if mount or format fails

Parameters

- conf: Pointer to *esp_vfs_spiffs_conf_t* configuration structure

esp_err_t **esp_vfs_spiffs_unregister** (**const** char *partition_label)

Unregister and unmount SPIFFS from VFS

Return

- ESP_OK if successful
- ESP_ERR_INVALID_STATE already unregistered

Parameters

- partition_label: Same label as passed to *esp_vfs_spiffs_register*.

bool **esp_spiffs_mounted** (**const** char *partition_label)

Check if SPIFFS is mounted

Return

- true if mounted
- false if not mounted

Parameters

- partition_label: Optional, label of the partition to check. If not specified, first partition with subtype=spiffs is used.

esp_err_t **esp_spiffs_format** (**const** char *partition_label)

Format the SPIFFS partition

Return

- ESP_OK if successful
- ESP_FAIL on error

Parameters

- partition_label: Same label as passed to *esp_vfs_spiffs_register*.

esp_err_t **esp_spiffs_info** (**const** char *partition_label, size_t *total_bytes, size_t *used_bytes)

Get information for SPIFFS

Return

- ESP_OK if success
- ESP_ERR_INVALID_STATE if not mounted

Parameters

- partition_label: Same label as passed to *esp_vfs_spiffs_register*
- [out] total_bytes: Size of the file system
- [out] used_bytes: Current used bytes in the file system

Structures

struct **esp_vfs_spiffs_conf_t**

Configuration structure for *esp_vfs_spiffs_register*.

Public Members

const char ***base_path**

File path prefix associated with the filesystem.

const char *partition_label

Optional, label of SPIFFS partition to use. If set to NULL, first partition with subtype=spiffs will be used.

size_t max_files

Maximum files that could be open at the same time.

bool format_if_mount_failed

If true, it will format the file system if it fails to mount.

2.6.8 Virtual filesystem component

Overview

Virtual filesystem (VFS) component provides a unified interface for drivers which can perform operations on file-like objects. These can be real filesystems (FAT, SPIFFS, etc.) or device drivers which provide a file-like interface.

This component allows C library functions, such as `fopen` and `fprintf`, to work with FS drivers. At a high level, each FS driver is associated with some path prefix. When one of C library functions needs to open a file, the VFS component searches for the FS driver associated with the file path and forwards the call to that driver. VFS also forwards read, write, and other calls for the given file to the same FS driver.

For example, one can register a FAT filesystem driver with the `/fat` prefix and call `fopen("/fat/file.txt", "w")`. The VFS component will then call the function `open` of the FAT driver and pass the argument `/file.txt` to it together with appropriate mode flags. All subsequent calls to C library functions for the returned `FILE*` stream will also be forwarded to the FAT driver.

FS registration

To register an FS driver, an application needs to define an instance of the `esp_vfs_t` structure and populate it with function pointers to FS APIs:

```
esp_vfs_t myfs = {
    .flags = ESP_VFS_FLAG_DEFAULT,
    .write = &myfs_write,
    .open = &myfs_open,
    .fstat = &myfs_fstat,
    .close = &myfs_close,
    .read = &myfs_read,
};

ESP_ERROR_CHECK(esp_vfs_register("/data", &myfs, NULL));
```

Depending on the way how the FS driver declares its API functions, either `read`, `write`, etc., or `read_p`, `write_p`, etc., should be used.

Case 1: API functions are declared without an extra context pointer (the FS driver is a singleton):

```
ssize_t myfs_write(int fd, const void * data, size_t size);

// In definition of esp_vfs_t:
    .flags = ESP_VFS_FLAG_DEFAULT,
    .write = &myfs_write,
// ... other members initialized

// When registering FS, context pointer (third argument) is NULL:
ESP_ERROR_CHECK(esp_vfs_register("/data", &myfs, NULL));
```

Case 2: API functions are declared with an extra context pointer (the FS driver supports multiple instances):

```

ssize_t myfs_write(myfs_t* fs, int fd, const void * data, size_t size);

// In definition of esp_vfs_t:
    .flags = ESP_VFS_FLAG_CONTEXT_PTR,
    .write_p = &myfs_write,
// ... other members initialized

// When registering FS, pass the FS context pointer into the third argument
// (hypothetical myfs_mount function is used for illustrative purposes)
myfs_t* myfs_inst1 = myfs_mount(partition1->offset, partition1->size);
ESP_ERROR_CHECK(esp_vfs_register("/data1", &myfs, myfs_inst1));

// Can register another instance:
myfs_t* myfs_inst2 = myfs_mount(partition2->offset, partition2->size);
ESP_ERROR_CHECK(esp_vfs_register("/data2", &myfs, myfs_inst2));

```

Synchronous input/output multiplexing Synchronous input/output multiplexing by `select()` is supported in the VFS component. The implementation works in the following way.

1. `select()` is called with file descriptors which could belong to various VFS drivers.
2. The file descriptors are divided into groups each belonging to one VFS driver.
3. The file descriptors belonging to non-socket VFS drivers are handed over to the given VFS drivers by `start_select()` described later on this page. This function represents the driver-specific implementation of `select()` for the given driver. This should be a non-blocking call which means the function should immediately return after setting up the environment for checking events related to the given file descriptors.
4. The file descriptors belonging to the socket VFS driver are handed over to the socket driver by `socket_select()` described later on this page. This is a blocking call which means that it will return only if there is an event related to socket file descriptors or a non-socket driver signals `socket_select()` to exit.
5. Results are collected from each VFS driver and all drivers are stopped by deinitialization of the environment for checking events.
6. The `select()` call ends and returns the appropriate results.

Non-socket VFS drivers If you want to use `select()` with a file descriptor belonging to a non-socket VFS driver then you need to register the driver with functions `start_select()` and `end_select()` similarly to the following example:

```

// In definition of esp_vfs_t:
    .start_select = &uart_start_select,
    .end_select = &uart_end_select,
// ... other members initialized

```

`start_select()` is called for setting up the environment for detection of read/write/error conditions on file descriptors belonging to the given VFS driver.

`end_select()` is called to stop/deinitialize/free the environment which was setup by `start_select()`.

Note: `end_select()` might be called without a previous `start_select()` call in some rare circumstances. `end_select()` should fail gracefully if this is the case.

Please refer to the reference implementation for the UART peripheral in `vfs/vfs_uart.c` and most particularly to the functions `esp_vfs_dev_uart_register()`, `uart_start_select()`, and `uart_end_select()` for more information.

Please check the following examples that demonstrate the use of `select()` with VFS file descriptors:

- [peripherals/uart/uart_select](#)
- [system/select](#)

Socket VFS drivers A socket VFS driver is using its own internal implementation of `select()` and non-socket VFS drivers notify it upon read/write/error conditions.

A socket VFS driver needs to be registered with the following functions defined:

```
// In definition of esp_vfs_t:
    .socket_select = &lwip_select,
    .get_socket_select_semaphore = &lwip_get_socket_select_semaphore,
    .stop_socket_select = &lwip_stop_socket_select,
    .stop_socket_select_isr = &lwip_stop_socket_select_isr,
// ... other members initialized
```

`socket_select()` is the internal implementation of `select()` for the socket driver. It works only with file descriptors belonging to the socket VFS.

`get_socket_select_semaphore()` returns the signalization object (semaphore) which will be used in non-socket drivers to stop the waiting in `socket_select()`.

`stop_socket_select()` call is used to stop the waiting in `socket_select()` by passing the object returned by `get_socket_select_semaphore()`.

`stop_socket_select_isr()` has the same functionality as `stop_socket_select()` but it can be used from ISR.

Please see [lwip/port/esp32/vfs_lwip.c](#) for a reference socket driver implementation using LWIP.

Note: If you use `select()` for socket file descriptors only then you can enable the `CONFIG_LWIP_USE_ONLY_LWIP_SELECT` option to reduce the code size and improve performance.

Note: Don't change the socket driver during an active `select()` call or you might experience some undefined behavior.

Paths

Each registered FS has a path prefix associated with it. This prefix can be considered as a “mount point” of this partition.

In case when mount points are nested, the mount point with the longest matching path prefix is used when opening the file. For instance, suppose that the following filesystems are registered in VFS:

- FS 1 on /data
- FS 2 on /data/static

Then:

- FS 1 will be used when opening a file called `/data/log.txt`
- FS 2 will be used when opening a file called `/data/static/index.html`
- Even if `/index.html` does not exist in FS 2, FS 1 will *not* be searched for `/static/index.html`.

As a general rule, mount point names must start with the path separator (`/`) and must contain at least one character after path separator. However, an empty mount point name is also supported and might be used in cases when an application needs to provide a “fallback” filesystem or to override VFS functionality altogether. Such filesystem will be used if no prefix matches the path given.

VFS does not handle dots (`.`) in path names in any special way. VFS does not treat `..` as a reference to the parent directory. In the above example, using a path `/data/static/../log.txt` will not result in a call to FS 1 to open `/log.txt`. Specific FS drivers (such as FATFS) might handle dots in file names differently.

When opening files, the FS driver receives only relative paths to files. For example:

1. The `myfs` driver is registered with `/data` as a path prefix.

2. The application calls `fopen("/data/config.json", ...)`.
3. The VFS component calls `myfs_open("/config.json", ...)`.
4. The `myfs` driver opens the `/config.json` file.

VFS does not impose any limit on total file path length, but it does limit the FS path prefix to `ESP_VFS_PATH_MAX` characters. Individual FS drivers may have their own filename length limitations.

File descriptors

File descriptors are small positive integers from 0 to `FD_SETSIZE - 1`, where `FD_SETSIZE` is defined in `newlib's sys/types.h`. The largest file descriptors (configured by `CONFIG_LWIP_MAX_SOCKETS`) are reserved for sockets. The VFS component contains a lookup-table called `s_fd_table` for mapping global file descriptors to VFS driver indexes registered in the `s_vfs` array.

Standard IO streams (stdin, stdout, stderr)

If the `menuconfig` option `UART for console output` is not set to `None`, then `stdin`, `stdout`, and `stderr` are configured to read from, and write to, a UART. It is possible to use `UART0` or `UART1` for standard IO. By default, `UART0` is used with 115200 baud rate; TX pin is `GPIO1`; RX pin is `GPIO3`. These parameters can be changed in `menuconfig`.

Writing to `stdout` or `stderr` will send characters to the UART transmit FIFO. Reading from `stdin` will retrieve characters from the UART receive FIFO.

By default, VFS uses simple functions for reading from and writing to UART. Writes busy-wait until all data is put into UART FIFO, and reads are non-blocking, returning only the data present in the FIFO. Due to this non-blocking read behavior, higher level C library calls, such as `fscanf("%d\n", &var);`, might not have desired results.

Applications which use the UART driver can instruct VFS to use the driver's interrupt driven, blocking read and write functions instead. This can be done using a call to the `esp_vfs_dev_uart_use_driver` function. It is also possible to revert to the basic non-blocking functions using a call to `esp_vfs_dev_uart_use_nonblocking`.

VFS also provides an optional newline conversion feature for input and output. Internally, most applications send and receive lines terminated by the LF (`'\n'`) character. Different terminal programs may require different line termination, such as CR or CRLF. Applications can configure this separately for input and output either via `menuconfig`, or by calls to the functions `esp_vfs_dev_uart_port_set_rx_line_endings` and `esp_vfs_dev_uart_port_set_tx_line_endings`.

Standard streams and FreeRTOS tasks `FILE` objects for `stdin`, `stdout`, and `stderr` are shared between all FreeRTOS tasks, but the pointers to these objects are stored in per-task `struct _reent`.

The following code is transferred to `fprintf(__getreent()->stderr, "42\n");` by the preprocessor:

```
fprintf(stderr, "42\n");
```

The `__getreent()` function returns a per-task pointer to `struct _reent` ([newlib/include/sys/reent.h#L370-L417](#)). This structure is allocated on the TCB of each task. When a task is initialized, `_stdin`, `_stdout`, and `_stderr` members of `struct _reent` are set to the values of `_stdin`, `_stdout`, and `_stderr` of `_GLOBAL_REENT` (i.e., the structure which is used before FreeRTOS is started).

Such a design has the following consequences:

- It is possible to set `stdin`, `stdout`, and `stderr` for any given task without affecting other tasks, e.g., by doing `stdin = fopen("/dev/uart/1", "r");`
- Closing default `stdin`, `stdout`, or `stderr` using `fclose` will close the `FILE` stream object, which will affect all other tasks.
- To change the default `stdin`, `stdout`, `stderr` streams for new tasks, modify `_GLOBAL_REENT->_stdin(_stdout, _stderr)` before creating the task.

Application Example

Instructions

API Reference

Header File

- `vfs/include/esp_vfs.h`

Functions

`ssize_t esp_vfs_write (struct _reent *r, int fd, const void *data, size_t size)`

These functions are to be used in newlib syscall table. They will be called by newlib when it needs to use any of the syscalls.

`off_t esp_vfs_lseek (struct _reent *r, int fd, off_t size, int mode)`

`ssize_t esp_vfs_read (struct _reent *r, int fd, void *dst, size_t size)`

`int esp_vfs_open (struct _reent *r, const char *path, int flags, int mode)`

`int esp_vfs_close (struct _reent *r, int fd)`

`int esp_vfs_fstat (struct _reent *r, int fd, struct stat *st)`

`int esp_vfs_stat (struct _reent *r, const char *path, struct stat *st)`

`int esp_vfs_link (struct _reent *r, const char *n1, const char *n2)`

`int esp_vfs_unlink (struct _reent *r, const char *path)`

`int esp_vfs_rename (struct _reent *r, const char *src, const char *dst)`

`int esp_vfs_utime (const char *path, const struct utimbuf *times)`

`esp_err_t esp_vfs_register (const char *base_path, const esp_vfs_t *vfs, void *ctx)`

Register a virtual filesystem for given path prefix.

Return ESP_OK if successful, ESP_ERR_NO_MEM if too many VFSEs are registered.

Parameters

- `base_path`: file path prefix associated with the filesystem. Must be a zero-terminated C string, up to ESP_VFS_PATH_MAX characters long, and at least 2 characters long. Name must start with a “/” and must not end with “/”. For example, “/data” or “/dev/spi” are valid. These VFSEs would then be called to handle file paths such as “/data/myfile.txt” or “/dev/spi/0”.
- `vfs`: Pointer to `esp_vfs_t`, a structure which maps syscalls to the filesystem driver functions. VFS component doesn't assume ownership of this pointer.
- `ctx`: If `vfs->flags` has ESP_VFS_FLAG_CONTEXT_PTR set, a pointer which should be passed to VFS functions. Otherwise, NULL.

`esp_err_t esp_vfs_register_fd_range (const esp_vfs_t *vfs, void *ctx, int min_fd, int max_fd)`

Special case function for registering a VFS that uses a method other than `open()` to open new file descriptors from the interval `<min_fd; max_fd)`.

This is a special-purpose function intended for registering LWIP sockets to VFS.

Return ESP_OK if successful, ESP_ERR_NO_MEM if too many VFSEs are registered, ESP_ERR_INVALID_ARG if the file descriptor boundaries are incorrect.

Parameters

- `vfs`: Pointer to `esp_vfs_t`. Meaning is the same as for `esp_vfs_register()`.
- `ctx`: Pointer to context structure. Meaning is the same as for `esp_vfs_register()`.
- `min_fd`: The smallest file descriptor this VFS will use.
- `max_fd`: Upper boundary for file descriptors this VFS will use (the biggest file descriptor plus one).

`esp_err_t esp_vfs_register_with_id (const esp_vfs_t *vfs, void *ctx, esp_vfs_id_t *vfs_id)`

Special case function for registering a VFS that uses a method other than `open()` to open new file descriptors. In

comparison with `esp_vfs_register_fd_range`, this function doesn't pre-registers an interval of file descriptors. File descriptors can be registered later, by using `esp_vfs_register_fd`.

Return `ESP_OK` if successful, `ESP_ERR_NO_MEM` if too many VFSes are registered, `ESP_ERR_INVALID_ARG` if the file descriptor boundaries are incorrect.

Parameters

- `vfs`: Pointer to `esp_vfs_t`. Meaning is the same as for `esp_vfs_register()`.
- `ctx`: Pointer to context structure. Meaning is the same as for `esp_vfs_register()`.
- `vfs_id`: Here will be written the VFS ID which can be passed to `esp_vfs_register_fd` for registering file descriptors.

esp_err_t **esp_vfs_unregister** (`const char *base_path`)

Unregister a virtual filesystem for given path prefix

Return `ESP_OK` if successful, `ESP_ERR_INVALID_STATE` if VFS for given prefix hasn't been registered

Parameters

- `base_path`: file prefix previously used in `esp_vfs_register` call

esp_err_t **esp_vfs_register_fd** (*esp_vfs_id_t* `vfs_id`, `int *fd`)

Special function for registering another file descriptor for a VFS registered by `esp_vfs_register_with_id`.

Return `ESP_OK` if the registration is successful, `ESP_ERR_NO_MEM` if too many file descriptors are registered, `ESP_ERR_INVALID_ARG` if the arguments are incorrect.

Parameters

- `vfs_id`: VFS identifier returned by `esp_vfs_register_with_id`.
- `fd`: The registered file descriptor will be written to this address.

esp_err_t **esp_vfs_unregister_fd** (*esp_vfs_id_t* `vfs_id`, `int fd`)

Special function for unregistering a file descriptor belonging to a VFS registered by `esp_vfs_register_with_id`.

Return `ESP_OK` if the registration is successful, `ESP_ERR_INVALID_ARG` if the arguments are incorrect.

Parameters

- `vfs_id`: VFS identifier returned by `esp_vfs_register_with_id`.
- `fd`: File descriptor which should be unregistered.

`int` **esp_vfs_select** (`int nfds`, `fd_set *readfds`, `fd_set *writefds`, `fd_set *errorfds`, `struct timeval *timeout`)

Synchronous I/O multiplexing which implements the functionality of POSIX `select()` for VFS.

Return The number of descriptors set in the descriptor sets, or -1 when an error (specified by `errno`) have occurred.

Parameters

- `nfds`: Specifies the range of descriptors which should be checked. The first `nfds` descriptors will be checked in each set.
- `readfds`: If not NULL, then points to a descriptor set that on input specifies which descriptors should be checked for being ready to read, and on output indicates which descriptors are ready to read.
- `writefds`: If not NULL, then points to a descriptor set that on input specifies which descriptors should be checked for being ready to write, and on output indicates which descriptors are ready to write.
- `errorfds`: If not NULL, then points to a descriptor set that on input specifies which descriptors should be checked for error conditions, and on output indicates which descriptors have error conditions.
- `timeout`: If not NULL, then points to `timeval` structure which specifies the time period after which the functions should time-out and return. If it is NULL, then the function will not time-out. Note that the timeout period is rounded up to the system tick and incremented by one.

`void` **esp_vfs_select_triggered** (*esp_vfs_select_sem_t* `sem`)

Notification from a VFS driver about a read/write/error condition.

This function is called when the VFS driver detects a read/write/error condition as it was requested by the previous call to `start_select`.

Parameters

- `sem`: semaphore structure which was passed to the driver by the `start_select` call

void **esp_vfs_select_triggered_isr** (*esp_vfs_select_sem_t sem*, BaseType_t **woken*)
Notification from a VFS driver about a read/write/error condition (ISR version)

This function is called when the VFS driver detects a read/write/error condition as it was requested by the previous call to `start_select`.

Parameters

- `sem`: semaphore structure which was passed to the driver by the `start_select` call
- `woken`: is set to `pdTRUE` if the function wakes up a task with higher priority

int **esp_vfs_poll** (**struct** pollfd **fds*, nfds_t *nfds*, int *timeout*)
Implements the VFS layer for synchronous I/O multiplexing by `poll()`

The implementation is based on `esp_vfs_select`. The parameters and return values are compatible with POSIX `poll()`.

Return A positive return value indicates the number of file descriptors that have been selected. The 0 return value indicates a timed-out poll. -1 is return on failure and `errno` is set accordingly.

Parameters

- `fds`: Pointer to the array containing file descriptors and events `poll()` should consider.
- `nfds`: Number of items in the array `fds`.
- `timeout`: `Poll()` should wait at least `timeout` milliseconds. If the value is 0 then it should return immediately. If the value is -1 then it should wait (block) until the event occurs.

ssize_t **esp_vfs_pread** (int *fd*, void **dst*, size_t *size*, off_t *offset*)
Implements the VFS layer of POSIX `pread()`

Return A positive return value indicates the number of bytes read. -1 is return on failure and `errno` is set accordingly.

Parameters

- `fd`: File descriptor used for read
- `dst`: Pointer to the buffer where the output will be written
- `size`: Number of bytes to be read
- `offset`: Starting offset of the read

ssize_t **esp_vfs_pwrite** (int *fd*, const void **src*, size_t *size*, off_t *offset*)
Implements the VFS layer of POSIX `pwrite()`

Return A positive return value indicates the number of bytes written. -1 is return on failure and `errno` is set accordingly.

Parameters

- `fd`: File descriptor used for write
- `src`: Pointer to the buffer from where the output will be read
- `size`: Number of bytes to write
- `offset`: Starting offset of the write

Structures

struct esp_vfs_select_sem_t
VFS semaphore type for `select()`

Public Members

bool **is_sem_local**
type of "sem" is `SemaphoreHandle_t` when true, defined by socket driver otherwise

void ***sem**
semaphore instance

struct esp_vfs_t
VFS definition structure.

This structure should be filled with pointers to corresponding FS driver functions.

VFS component will translate all FDs so that the filesystem implementation sees them starting at zero. The caller sees a global FD which is prefixed with an pre-filesystem-implementation.

Some FS implementations expect some state (e.g. pointer to some structure) to be passed in as a first argument. For these implementations, populate the members of this structure which have `_p` suffix, set flags member to `ESP_VFS_FLAG_CONTEXT_PTR` and provide the context pointer to `esp_vfs_register` function. If the implementation doesn't use this extra argument, populate the members without `_p` suffix and set flags member to `ESP_VFS_FLAG_DEFAULT`.

If the FS driver doesn't provide some of the functions, set corresponding members to `NULL`.

Public Members

int **flags**

`ESP_VFS_FLAG_CONTEXT_PTR` or `ESP_VFS_FLAG_DEFAULT`

esp_err_t (***start_select**) (int nfd, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, *esp_vfs_select_sem_t* sem, void **end_select_args)

start_select is called for setting up synchronous I/O multiplexing of the desired file descriptors in the given VFS

int (***socket_select**) (int nfd, fd_set *readfds, fd_set *writefds, fd_set *errorfds, **struct** timeval *timeout)

socket select function for socket FDs with the functionality of POSIX select(); this should be set only for the socket VFS

void (***stop_socket_select**) (void *sem)

called by VFS to interrupt the socket_select call when select is activated from a non-socket VFS driver; set only for the socket driver

void (***stop_socket_select_isr**) (void *sem, BaseType_t *woken)

stop_socket_select which can be called from ISR; set only for the socket driver

void (***get_socket_select_semaphore**) (void)

end_select is called to stop the I/O multiplexing and deinitialize the environment created by start_select for the given VFS

esp_err_t (***end_select**) (void *end_select_args)

get_socket_select_semaphore returns semaphore allocated in the socket driver; set only for the socket driver

Macros

MAX_FDS

Maximum number of (global) file descriptors.

ESP_VFS_PATH_MAX

Maximum length of path prefix (not including zero terminator)

ESP_VFS_FLAG_DEFAULT

Default value of flags member in *esp_vfs_t* structure.

ESP_VFS_FLAG_CONTEXT_PTR

Flag which indicates that FS needs extra context pointer in syscalls.

Type Definitions

typedef int **esp_vfs_id_t**

Header File

- [vfs/include/esp_vfs_dev.h](#)

Functions

void **esp_vfs_dev_uart_register** (void)
add /dev/uart virtual filesystem driver

This function is called from startup code to enable serial output

void **esp_vfs_dev_uart_set_rx_line_endings** (*esp_line_endings_t mode*)
Set the line endings expected to be received on UART.

This specifies the conversion between line endings received on UART and newlines (‘ ’ , LF) passed into stdin:

- ESP_LINE_ENDINGS_CRLF: convert CRLF to LF
- ESP_LINE_ENDINGS_CR: convert CR to LF
- ESP_LINE_ENDINGS_LF: no modification

Note this function is not thread safe w.r.t. reading from UART

Parameters

- mode: line endings expected on UART

void **esp_vfs_dev_uart_set_tx_line_endings** (*esp_line_endings_t mode*)
Set the line endings to sent to UART.

This specifies the conversion between newlines (‘ ’ , LF) on stdout and line endings sent over UART:

- ESP_LINE_ENDINGS_CRLF: convert LF to CRLF
- ESP_LINE_ENDINGS_CR: convert LF to CR
- ESP_LINE_ENDINGS_LF: no modification

Note this function is not thread safe w.r.t. writing to UART

Parameters

- mode: line endings to send to UART

int **esp_vfs_dev_uart_port_set_rx_line_endings** (int *uart_num*, *esp_line_endings_t mode*)
Set the line endings expected to be received on specified UART.

This specifies the conversion between line endings received on UART and newlines (‘ ’ , LF) passed into stdin:

- ESP_LINE_ENDINGS_CRLF: convert CRLF to LF
- ESP_LINE_ENDINGS_CR: convert CR to LF
- ESP_LINE_ENDINGS_LF: no modification

Note this function is not thread safe w.r.t. reading from UART

Return 0 if succeeded, or -1 when an error (specified by errno) have occurred.

Parameters

- *uart_num*: the UART number
- mode: line endings to send to UART

int **esp_vfs_dev_uart_port_set_tx_line_endings** (int *uart_num*, *esp_line_endings_t mode*)
Set the line endings to sent to specified UART.

This specifies the conversion between newlines (‘ ’ , LF) on stdout and line endings sent over UART:

- ESP_LINE_ENDINGS_CRLF: convert LF to CRLF
- ESP_LINE_ENDINGS_CR: convert LF to CR
- ESP_LINE_ENDINGS_LF: no modification

Note this function is not thread safe w.r.t. writing to UART

Return 0 if succeeded, or -1 when an error (specified by errno) have occurred.

Parameters

- *uart_num*: the UART number
- mode: line endings to send to UART

void **esp_vfs_dev_uart_use_nonblocking** (int *uart_num*)
set VFS to use simple functions for reading and writing UART Read is non-blocking, write is busy waiting until TX FIFO has enough space. These functions are used by default.

Parameters

- `uart_num`: UART peripheral number

void `esp_vfs_dev_uart_use_driver` (int `uart_num`)
set VFS to use UART driver for reading and writing

Note application must configure UART driver before calling these functions. With these functions, read and write are blocking and interrupt-driven.

Parameters

- `uart_num`: UART peripheral number

Enumerations

enum `esp_line_endings_t`

Line ending settings.

Values:

`ESP_LINE_ENDINGS_CRLF`
CR + LF.

`ESP_LINE_ENDINGS_CR`
CR.

`ESP_LINE_ENDINGS_LF`
LF.

2.6.9 Wear Levelling API

Overview

Most of flash memory and especially SPI flash that is used in ESP32 has a sector-based organization and also has a limited number of erase/modification cycles per memory sector. The wear levelling component helps to distribute wear and tear among sectors more evenly without requiring any attention from the user.

The wear levelling component provides API functions related to reading, writing, erasing, and memory mapping of data in external SPI flash through the partition component. The component also has higher-level API functions which work with the FAT filesystem defined in *FAT filesystem*.

The wear levelling component, together with the FAT FS component, uses FAT FS sectors of 4096 bytes, which is a standard size for flash memory. With this size, the component shows the best performance but needs additional memory in RAM.

To save internal memory, the component has two additional modes which both use sectors of 512 bytes:

- **Performance mode.** Erase sector operation data is stored in RAM, the sector is erased, and then data is copied back to flash memory. However, if a device is powered off for any reason, all 4096 bytes of data is lost.
- **Safety mode.** The data is first saved to flash memory, and after the sector is erased, the data is saved back. If a device is powered off, the data can be recovered as soon as the device boots up.

The default settings are as follows: - Sector size is 512 bytes - Performance mode

You can change the settings through the configuration menu.

The wear levelling component does not cache data in RAM. The write and erase functions modify flash directly, and flash contents are consistent when the function returns.

Wear Levelling access API functions

This is the set of API functions for working with data in flash:

- `wl_mount` - initializes the wear levelling module and mounts the specified partition
- `wl_unmount` - unmounts the partition and deinitializes the wear levelling module
- `wl_erase_range` - erases a range of addresses in flash

- `wl_write` - writes data to a partition
- `wl_read` - reads data from a partition
- `wl_size` - returns the size of available memory in bytes
- `wl_sector_size` - returns the size of one sector

As a rule, try to avoid using raw wear levelling functions and use filesystem-specific functions instead.

Memory Size

The memory size is calculated in the wear levelling module based on partition parameters. The module uses some sectors of flash for internal data.

See also

- [FAT Filesystem](#)
- [Partition Table documentation](#)

Application Example

An example which combines the wear levelling driver with the FATFS library is provided in the [storage/wear_levelling](#) directory. This example initializes the wear levelling driver, mounts FATFS partition, as well as writes and reads data from it using POSIX and C library APIs. See the [storage/wear_levelling/README.md](#) file for more information.

High level API Reference

Header Files

- [fatfs/src/esp_vfs_fat.h](#)

Functions

`esp_err_t esp_vfs_fat_spiflash_mount` (`const` char **base_path*, `const` char **partition_label*,
 `const` [esp_vfs_fat_mount_config_t](#) **mount_config*,
 [wl_handle_t](#) **wl_handle*)

Convenience function to initialize FAT filesystem in SPI flash and register it in VFS.

This is an all-in-one function which does the following:

- finds the partition with defined `partition_label`. Partition label should be configured in the partition table.
- initializes flash wear levelling library on top of the given partition
- mounts FAT partition using FATFS library on top of flash wear levelling library
- registers FATFS library with VFS, with prefix given by `base_prefix` variable

This function is intended to make example code more compact.

Return

- `ESP_OK` on success
- `ESP_ERR_NOT_FOUND` if the partition table does not contain FATFS partition with given label
- `ESP_ERR_INVALID_STATE` if `esp_vfs_fat_spiflash_mount` was already called
- `ESP_ERR_NO_MEM` if memory can not be allocated
- `ESP_FAIL` if partition can not be mounted
- other error codes from wear levelling library, SPI flash driver, or FATFS drivers

Parameters

- `base_path`: path where FATFS partition should be mounted (e.g. `"/spiflash"`)
- `partition_label`: label of the partition which should be used
- `mount_config`: pointer to structure with extra parameters for mounting FATFS
- [`out`] `wl_handle`: wear levelling driver handle

struct `esp_vfs_fat_mount_config_t`

Configuration arguments for `esp_vfs_fat_sdmmc_mount` and `esp_vfs_fat_spiflash_mount` functions.

Public Members

bool **format_if_mount_failed**

If FAT partition can not be mounted, and this parameter is true, create partition table and format the filesystem.

int **max_files**

Max number of open files.

size_t **allocation_unit_size**

If `format_if_mount_failed` is set, and mount fails, format the card with given allocation unit size. Must be a power of 2, between sector size and $128 * \text{sector size}$. For SD cards, sector size is always 512 bytes. For wear_levelling, sector size is determined by `CONFIG_WL_SECTOR_SIZE` option.

Using larger allocation unit size will result in higher read/write performance and higher overhead when storing small files.

Setting this field to 0 will result in allocation unit set to the sector size.

esp_err_t **esp_vfs_fat_spiflash_unmount** (**const** char **base_path*, *wl_handle_t* *wl_handle*)

Unmount FAT filesystem and release resources acquired using `esp_vfs_fat_spiflash_mount`.

Return

- ESP_OK on success
- ESP_ERR_INVALID_STATE if `esp_vfs_fat_spiflash_mount` hasn't been called

Parameters

- *base_path*: path where partition should be registered (e.g. `"/spiflash"`)
- *wl_handle*: wear levelling driver handle returned by `esp_vfs_fat_spiflash_mount`

Mid level API Reference

Header File

- [wear_levelling/include/wear_levelling.h](#)

Functions

esp_err_t **wl_mount** (**const** *esp_partition_t* **partition*, *wl_handle_t* **out_handle*)

Mount WL for defined partition.

Return

- ESP_OK, if the allocation was successfully;
- ESP_ERR_INVALID_ARG, if WL allocation was unsuccessful;
- ESP_ERR_NO_MEM, if there was no memory to allocate WL components;

Parameters

- *partition*: that will be used for access
- *out_handle*: handle of the WL instance

esp_err_t **wl_unmount** (*wl_handle_t* *handle*)

Unmount WL for defined partition.

Return

- ESP_OK, if the operation completed successfully;
- or one of error codes from lower-level flash driver.

Parameters

- *handle*: WL partition handle

esp_err_t **wl_erase_range** (*wl_handle_t* *handle*, size_t *start_addr*, size_t *size*)

Erase part of the WL storage.

Return

- ESP_OK, if the range was erased successfully;
- ESP_ERR_INVALID_ARG, if iterator or dst are NULL;
- ESP_ERR_INVALID_SIZE, if erase would go out of bounds of the partition;

- or one of error codes from lower-level flash driver.

Parameters

- `handle`: WL handle that are related to the partition
- `start_addr`: Address where erase operation should start. Must be aligned to the result of function `wl_sector_size(...)`.
- `size`: Size of the range which should be erased, in bytes. Must be divisible by result of function `wl_sector_size(...)`.

esp_err_t **wl_write** (*wl_handle_t* handle, *size_t* dest_addr, **const** void *src, *size_t* size)

Write data to the WL storage.

Before writing data to flash, corresponding region of flash needs to be erased. This can be done using `wl_erase_range` function.

Note Prior to writing to WL storage, make sure it has been erased with `wl_erase_range` call.

Return

- `ESP_OK`, if data was written successfully;
- `ESP_ERR_INVALID_ARG`, if `dst_offset` exceeds partition size;
- `ESP_ERR_INVALID_SIZE`, if write would go out of bounds of the partition;
- or one of error codes from lower-level flash driver.

Parameters

- `handle`: WL handle that are related to the partition
- `dest_addr`: Address where the data should be written, relative to the beginning of the partition.
- `src`: Pointer to the source buffer. Pointer must be non-NULL and buffer must be at least 'size' bytes long.
- `size`: Size of data to be written, in bytes.

esp_err_t **wl_read** (*wl_handle_t* handle, *size_t* src_addr, void *dest, *size_t* size)

Read data from the WL storage.

Return

- `ESP_OK`, if data was read successfully;
- `ESP_ERR_INVALID_ARG`, if `src_offset` exceeds partition size;
- `ESP_ERR_INVALID_SIZE`, if read would go out of bounds of the partition;
- or one of error codes from lower-level flash driver.

Parameters

- `handle`: WL module instance that was initialized before
- `dest`: Pointer to the buffer where data should be stored. Pointer must be non-NULL and buffer must be at least 'size' bytes long.
- `src_addr`: Address of the data to be read, relative to the beginning of the partition.
- `size`: Size of data to be read, in bytes.

size_t **wl_size** (*wl_handle_t* handle)

Get size of the WL storage.

Return usable size, in bytes

Parameters

- `handle`: WL module handle that was initialized before

size_t **wl_sector_size** (*wl_handle_t* handle)

Get sector size of the WL instance.

Return sector size, in bytes

Parameters

- `handle`: WL module handle that was initialized before

Macros

WL_INVALID_HANDLE

Type Definitions

```
typedef int32_t wl_handle_t
    wear levelling handle
```

Code examples for this API section are provided in the [storage](#) directory of ESP-IDF examples.

2.7 System API

2.7.1 App Image Format

An application image consists of the following structures:

1. The `esp_image_header_t` structure describes the mode of SPI flash and the count of memory segments.
2. The `esp_image_segment_header_t` structure describes each segment, its length, and its location in ESP32's memory, followed by the data with a length of `data_len`. The data offset for each segment in the image is calculated in the following way:
 - offset for 0 Segment = `sizeof(esp_image_header_t) + sizeof(esp_image_segment_header_t)`.
 - offset for 1 Segment = offset for 0 Segment + length of 0 Segment + `sizeof(esp_image_segment_header_t)`.
 - offset for 2 Segment = offset for 1 Segment + length of 1 Segment + `sizeof(esp_image_segment_header_t)`.
 - ...

The count of each segment is defined in the `segment_count` field that is stored in `esp_image_header_t`. The count cannot be more than `ESP_IMAGE_MAX_SEGMENTS`.

To get the list of your image segments, please run the following command:

```
esptool.py --chip esp32 image_info build/app.bin
```

```
esptool.py v2.3.1
Image version: 1
Entry point: 40080ea4
13 segments
Segment 1: len 0x13ce0 load 0x3f400020 file_offs 0x00000018 SOC_DROM
Segment 2: len 0x00000 load 0x3ff80000 file_offs 0x00013d00 SOC_RTC_DRAM
Segment 3: len 0x00000 load 0x3ff80000 file_offs 0x00013d08 SOC_RTC_DRAM
Segment 4: len 0x028e0 load 0x3ffb0000 file_offs 0x00013d10 DRAM
Segment 5: len 0x00000 load 0x3ffb28e0 file_offs 0x000165f8 DRAM
Segment 6: len 0x00400 load 0x40080000 file_offs 0x00016600 SOC_IRAM
Segment 7: len 0x09600 load 0x40080400 file_offs 0x00016a08 SOC_IRAM
Segment 8: len 0x62e4c load 0x400d0018 file_offs 0x00020010 SOC_IROM
Segment 9: len 0x06cec load 0x40089a00 file_offs 0x00082e64 SOC_IROM
Segment 10: len 0x00000 load 0x400c0000 file_offs 0x00089b58 SOC_RTC_IRAM
Segment 11: len 0x00004 load 0x50000000 file_offs 0x00089b60 SOC_RTC_DATA
Segment 12: len 0x00000 load 0x50000004 file_offs 0x00089b6c SOC_RTC_DATA
Segment 13: len 0x00000 load 0x50000004 file_offs 0x00089b74 SOC_RTC_DATA
Checksum: e8 (valid) Validation Hash: ↪
↪407089ca0eae2bbf83b4120979d3354b1c938a49cb7a0c997f240474ef2ec76b (valid)
```

You can also see the information on segments in the IDF logs while your application is booting:

```
I (443) esp_image: segment 0: paddr=0x00020020 vaddr=0x3f400020 size=0x13ce0 ( ↪
↪81120) map
I (489) esp_image: segment 1: paddr=0x00033d08 vaddr=0x3ff80000 size=0x00000 ( 0) ↪
↪load
I (530) esp_image: segment 2: paddr=0x00033d10 vaddr=0x3ff80000 size=0x00000 ( 0) ↪
↪load
I (571) esp_image: segment 3: paddr=0x00033d18 vaddr=0x3ffb0000 size=0x028e0 ( ↪
↪10464) load
I (612) esp_image: segment 4: paddr=0x00036600 vaddr=0x3ffb28e0 size=0x00000 ( 0) ↪
↪load
```

(continues on next page)

(continued from previous page)

```

I (654) esp_image: segment 5: paddr=0x00036608 vaddr=0x40080000 size=0x00400 ( 1)
↳1024) load
I (695) esp_image: segment 6: paddr=0x00036a10 vaddr=0x40080400 size=0x09600 ( 1)
↳38400) load
I (737) esp_image: segment 7: paddr=0x00040018 vaddr=0x400d0018 size=0x62e4c ( 1)
↳405068) map
I (847) esp_image: segment 8: paddr=0x000a2e6c vaddr=0x40089a00 size=0x06cec ( 1)
↳27884) load
I (888) esp_image: segment 9: paddr=0x000a9b60 vaddr=0x400c0000 size=0x00000 ( 0)
↳load
I (929) esp_image: segment 10: paddr=0x000a9b68 vaddr=0x50000000 size=0x00004 ( 4)
↳load
I (971) esp_image: segment 11: paddr=0x000a9b74 vaddr=0x50000004 size=0x00000 ( 0)
↳load
I (1012) esp_image: segment 12: paddr=0x000a9b7c vaddr=0x50000004 size=0x00000 ( 0)
↳0) load

```

For more details on the type of memory segments and their address ranges, see the ESP32 Technical Reference Manual, Section 1.3.2 *Embedded Memory*.

3. The image has a single checksum byte after the last segment. This byte is written on a sixteen byte padded boundary, so the application image might need padding.
4. If the `hash_appended` field from `esp_image_header_t` is set then a SHA256 checksum will be appended. The value of SHA256 is calculated on the range from first byte and up to this field. The length of this field is 32 bytes.
5. If the options `CONFIG_SECURE_SIGNED_APPS_SCHEME` is set to ECDSA then the application image will have additional 68 bytes for an ECDSA signature, which includes:
 - version word (4 bytes),
 - signature data (64 bytes).

Application Description

The DROM segment starts with the `esp_app_desc_t` structure which carries specific fields describing the application:

- `secure_version` - see [Anti-rollback](#).
- `version` - see [App version](#). *
- `project_name` is filled from `PROJECT_NAME`. *
- `time and date` - compile time and date.
- `idf_ver` - version of ESP-IDF. *
- `app_elf_sha256` - contains sha256 for the elf application file.

* - The maximum length is 32 characters, including null-termination character. For example, if the length of `PROJECT_NAME` exceeds 32 characters, the excess characters will be disregarded.

This structure is useful for identification of images uploaded OTA because it has a fixed offset = `sizeof(esp_image_header_t) + sizeof(esp_image_segment_header_t)`. As soon as a device receives the first fragment containing this structure, it has all the information to determine whether the update should be continued or not.

Adding a Custom Structure to an Application

Customer also has the opportunity to have similar structure with a fixed offset relative to the beginning of the image. The following pattern can be used to add a custom structure to your image:

```

const __attribute__((section(".rodata_custom_desc"))) esp_custom_app_desc_t custom_
↳app_desc = { ... }

```

Offset for custom structure is `sizeof(esp_image_header_t) + sizeof(esp_image_segment_header_t) + sizeof(esp_app_desc_t)`.

To guarantee that the custom structure is located in the image even if it is not used, you need to add:

- For Make: add `COMPONENT_ADD_LDFLAGS += -u custom_app_desc` into `component.mk`
- For Cmake: add `target_link_libraries(${COMPONENT_TARGET} "-u custom_app_desc")` into `CMakeLists.txt`

API Reference

Header File

- [bootloader_support/include/esp_app_format.h](#)

Structures

struct esp_image_header_t

Main header of binary image.

Public Members

uint8_t magic

Magic word `ESP_IMAGE_HEADER_MAGIC`

uint8_t segment_count

Count of memory segments

uint8_t spi_mode

flash read mode (`esp_image_spi_mode_t` as `uint8_t`)

uint8_t spi_speed : 4

flash frequency (`esp_image_spi_freq_t` as `uint8_t`)

uint8_t spi_size : 4

flash chip size (`esp_image_flash_size_t` as `uint8_t`)

uint32_t entry_addr

Entry address

uint8_t wp_pin

WP pin when SPI pins set via efuse (read by ROM bootloader, the IDF bootloader uses software to configure the WP pin and sets this field to `0xEE`=disabled)

uint8_t spi_pin_drv[3]

Drive settings for the SPI flash pins (read by ROM bootloader)

esp_chip_id_t chip_id

Chip identification number

uint8_t min_chip_rev

Minimum chip revision supported by image

uint8_t reserved[8]

Reserved bytes in additional header space, currently unused

uint8_t hash_appended

If 1, a SHA256 digest “simple hash” (of the entire image) is appended after the checksum. Included in image length. This digest is separate to secure boot and only used for detecting corruption. For secure boot signed images, the signature is appended after this (and the simple hash is included in the signed data).

struct esp_image_segment_header_t

Header of binary image segment.

Public Members

`uint32_t load_addr`
Address of segment

`uint32_t data_len`
Length of data

`struct esp_app_desc_t`
Description about application.

Public Members

`uint32_t magic_word`
Magic word `ESP_APP_DESC_MAGIC_WORD`

`uint32_t secure_version`
Secure version

`uint32_t reserv1[2]`
reserv1

`char version[32]`
Application version

`char project_name[32]`
Project name

`char time[16]`
Compile time

`char date[16]`
Compile date

`char idf_ver[32]`
Version IDF

`uint8_t app_elf_sha256[32]`
sha256 of elf file

`uint32_t reserv2[20]`
reserv2

Macros

`ESP_IMAGE_HEADER_MAGIC`
The magic word for the *esp_image_header_t* structure.

`ESP_IMAGE_MAX_SEGMENTS`
Max count of segments in the image.

`ESP_APP_DESC_MAGIC_WORD`
The magic word for the `esp_app_desc` structure that is in DROM.

Enumerations

`enum esp_chip_id_t`
ESP chip ID.

Values:

`ESP_CHIP_ID_ESP32 = 0x0000`
chip ID: ESP32

`ESP_CHIP_ID_INVALID = 0xFFFF`
Invalid chip ID (we defined it to make sure the `esp_chip_id_t` is 2 bytes size)

enum esp_image_spi_mode_t
SPI flash mode, used in [esp_image_header_t](#).

Values:

ESP_IMAGE_SPI_MODE_QIO
SPI mode QIO

ESP_IMAGE_SPI_MODE_QOUT
SPI mode QOUT

ESP_IMAGE_SPI_MODE_DIO
SPI mode DIO

ESP_IMAGE_SPI_MODE_DOUT
SPI mode DOUT

ESP_IMAGE_SPI_MODE_FAST_READ
SPI mode FAST_READ

ESP_IMAGE_SPI_MODE_SLOW_READ
SPI mode SLOW_READ

enum esp_image_spi_freq_t
SPI flash clock frequency.

Values:

ESP_IMAGE_SPI_SPEED_40M
SPI clock frequency 40 MHz

ESP_IMAGE_SPI_SPEED_26M
SPI clock frequency 26 MHz

ESP_IMAGE_SPI_SPEED_20M
SPI clock frequency 20 MHz

ESP_IMAGE_SPI_SPEED_80M = 0xF
SPI clock frequency 80 MHz

enum esp_image_flash_size_t
Supported SPI flash sizes.

Values:

ESP_IMAGE_FLASH_SIZE_1MB = 0
SPI flash size 1 MB

ESP_IMAGE_FLASH_SIZE_2MB
SPI flash size 2 MB

ESP_IMAGE_FLASH_SIZE_4MB
SPI flash size 4 MB

ESP_IMAGE_FLASH_SIZE_8MB
SPI flash size 8 MB

ESP_IMAGE_FLASH_SIZE_16MB
SPI flash size 16 MB

ESP_IMAGE_FLASH_SIZE_MAX
SPI flash size MAX

2.7.2 Application Level Tracing

Overview

IDF provides useful feature for program behaviour analysis: application level tracing. It is implemented in the corresponding library and can be enabled via menuconfig. This feature allows to transfer arbitrary data between host and ESP32 via JTAG interface with small overhead on program execution. Developers can use this library to send application specific state of execution to the host and receive commands or other type of info in the opposite direction at runtime. The main use cases of this library are:

1. Collecting application specific data, see [Application Specific Tracing](#)
2. Lightweight logging to the host, see [Logging to Host](#)
3. System behaviour analysis, see [System Behavior Analysis with SEGGER System View](#)

API Reference

Header File

- `app_trace/include/esp_app_trace.h`

Functions

`esp_err_t esp_apptrace_init` (void)

Initializes application tracing module.

Note Should be called before any `esp_apptrace_XXX` call.

Return ESP_OK on success, otherwise see `esp_err_t`

void `esp_apptrace_down_buffer_config` (uint8_t *buf, uint32_t size)

Configures down buffer.

Note Needs to be called before initiating any data transfer using `esp_apptrace_buffer_get` and `esp_apptrace_write`. This function does not protect internal data by lock.

Parameters

- `buf`: Address of buffer to use for down channel (host to target) data.
- `size`: Size of the buffer.

uint8_t *`esp_apptrace_buffer_get` (`esp_apptrace_dest_t dest`, uint32_t size, uint32_t tmo)

Allocates buffer for trace data. After data in buffer are ready to be sent off `esp_apptrace_buffer_put` must be called to indicate it.

Return non-NULL on success, otherwise NULL.

Parameters

- `dest`: Indicates HW interface to send data.
- `size`: Size of data to write to trace buffer.
- `tmo`: Timeout for operation (in us). Use `ESP_APPTRACE_TMO_INFINITE` to wait indefinitely.

`esp_err_t esp_apptrace_buffer_put` (`esp_apptrace_dest_t dest`, uint8_t *ptr, uint32_t tmo)

Indicates that the data in buffer are ready to be sent off. This function is a counterpart of and must be preceded by `esp_apptrace_buffer_get`.

Return ESP_OK on success, otherwise see `esp_err_t`

Parameters

- `dest`: Indicates HW interface to send data. Should be identical to the same parameter in call to `esp_apptrace_buffer_get`.
- `ptr`: Address of trace buffer to release. Should be the value returned by call to `esp_apptrace_buffer_get`.
- `tmo`: Timeout for operation (in us). Use `ESP_APPTRACE_TMO_INFINITE` to wait indefinitely.

`esp_err_t esp_apptrace_write` (`esp_apptrace_dest_t dest`, const void *data, uint32_t size, uint32_t tmo)

Writes data to trace buffer.

Return ESP_OK on success, otherwise see `esp_err_t`

Parameters

- `dest`: Indicates HW interface to send data.

- `data`: Address of data to write to trace buffer.
- `size`: Size of data to write to trace buffer.
- `tmo`: Timeout for operation (in us). Use `ESP_APPTRACE_TMO_INFINITE` to wait indefinitely.

int **esp_apptrace_vprintf_to** (*esp_apptrace_dest_t* `dest`, *uint32_t* `tmo`, **const** char **fmt*, *va_list* `ap`)
vprintf-like function to sent log messages to host via specified HW interface.

Return Number of bytes written.

Parameters

- `dest`: Indicates HW interface to send data.
- `tmo`: Timeout for operation (in us). Use `ESP_APPTRACE_TMO_INFINITE` to wait indefinitely.
- `fmt`: Address of format string.
- `ap`: List of arguments.

int **esp_apptrace_vprintf** (**const** char **fmt*, *va_list* `ap`)
vprintf-like function to sent log messages to host.

Return Number of bytes written.

Parameters

- `fmt`: Address of format string.
- `ap`: List of arguments.

esp_err_t **esp_apptrace_flush** (*esp_apptrace_dest_t* `dest`, *uint32_t* `tmo`)
Flushes remaining data in trace buffer to host.

Return `ESP_OK` on success, otherwise see `esp_err_t`

Parameters

- `dest`: Indicates HW interface to flush data on.
- `tmo`: Timeout for operation (in us). Use `ESP_APPTRACE_TMO_INFINITE` to wait indefinitely.

esp_err_t **esp_apptrace_flush_nolock** (*esp_apptrace_dest_t* `dest`, *uint32_t* `min_sz`, *uint32_t* `tmo`)
Flushes remaining data in trace buffer to host without locking internal data. This is special version of `esp_apptrace_flush` which should be called from panic handler.

Return `ESP_OK` on success, otherwise see `esp_err_t`

Parameters

- `dest`: Indicates HW interface to flush data on.
- `min_sz`: Threshold for flushing data. If current filling level is above this value, data will be flushed. TRAX destinations only.
- `tmo`: Timeout for operation (in us). Use `ESP_APPTRACE_TMO_INFINITE` to wait indefinitely.

esp_err_t **esp_apptrace_read** (*esp_apptrace_dest_t* `dest`, void **data*, *uint32_t* **size*, *uint32_t* `tmo`)
Reads host data from trace buffer.

Return `ESP_OK` on success, otherwise see `esp_err_t`

Parameters

- `dest`: Indicates HW interface to read the data on.
- `data`: Address of buffer to put data from trace buffer.
- `size`: Pointer to store size of read data. Before call to this function pointed memory must hold requested size of data
- `tmo`: Timeout for operation (in us). Use `ESP_APPTRACE_TMO_INFINITE` to wait indefinitely.

uint8_t ***esp_apptrace_down_buffer_get** (*esp_apptrace_dest_t* `dest`, *uint32_t* **size*, *uint32_t* `tmo`)
Retrieves incoming data buffer if any. After data in buffer are processed `esp_apptrace_down_buffer_put` must be called to indicate it.

Return non-NULL on success, otherwise NULL.

Parameters

- `dest`: Indicates HW interface to receive data.
- `size`: Address to store size of available data in down buffer. Must be initialized with requested value.
- `tmo`: Timeout for operation (in us). Use `ESP_APPTRACE_TMO_INFINITE` to wait indefinitely.

esp_err_t **esp_apprtrace_down_buffer_put** (*esp_apprtrace_dest_t* dest, uint8_t *ptr, uint32_t tmo)

Indicates that the data in down buffer are processed. This function is a counterpart of and must be preceded by `esp_apprtrace_down_buffer_get`.

Return ESP_OK on success, otherwise see `esp_err_t`

Parameters

- `dest`: Indicates HW interface to receive data. Should be identical to the same parameter in call to `esp_apprtrace_down_buffer_get`.
- `ptr`: Address of trace buffer to release. Should be the value returned by call to `esp_apprtrace_down_buffer_get`.
- `tmo`: Timeout for operation (in us). Use `ESP_APPTRACE_TMO_INFINITE` to wait indefinitely.

bool **esp_apprtrace_host_is_connected** (*esp_apprtrace_dest_t* dest)

Checks whether host is connected.

Return true if host is connected, otherwise false

Parameters

- `dest`: Indicates HW interface to use.

void ***esp_apprtrace_fopen** (*esp_apprtrace_dest_t* dest, const char *path, const char *mode)

Opens file on host. This function has the same semantic as 'fopen' except for the first argument.

Return non zero file handle on success, otherwise 0

Parameters

- `dest`: Indicates HW interface to use.
- `path`: Path to file.
- `mode`: Mode string. See `fopen` for details.

int **esp_apprtrace_fclose** (*esp_apprtrace_dest_t* dest, void *stream)

Closes file on host. This function has the same semantic as 'fclose' except for the first argument.

Return Zero on success, otherwise non-zero. See `fclose` for details.

Parameters

- `dest`: Indicates HW interface to use.
- `stream`: File handle returned by `esp_apprtrace_fopen`.

size_t **esp_apprtrace_fwrite** (*esp_apprtrace_dest_t* dest, const void *ptr, size_t size, size_t nmemb, void *stream)

Writes to file on host. This function has the same semantic as 'fwrite' except for the first argument.

Return Number of written items. See `fwrite` for details.

Parameters

- `dest`: Indicates HW interface to use.
- `ptr`: Address of data to write.
- `size`: Size of an item.
- `nmemb`: Number of items to write.
- `stream`: File handle returned by `esp_apprtrace_fopen`.

size_t **esp_apprtrace_fread** (*esp_apprtrace_dest_t* dest, void *ptr, size_t size, size_t nmemb, void *stream)

Read file on host. This function has the same semantic as 'fread' except for the first argument.

Return Number of read items. See `fread` for details.

Parameters

- `dest`: Indicates HW interface to use.
- `ptr`: Address to store read data.
- `size`: Size of an item.
- `nmemb`: Number of items to read.
- `stream`: File handle returned by `esp_apprtrace_fopen`.

int **esp_apprtrace_fseek** (*esp_apprtrace_dest_t* dest, void *stream, long offset, int whence)

Set position indicator in file on host. This function has the same semantic as 'fseek' except for the first argument.

Return Zero on success, otherwise non-zero. See `fseek` for details.

Parameters

- `dest`: Indicates HW interface to use.
- `stream`: File handle returned by `esp_appttrace_fopen`.
- `offset`: Offset. See `fseek` for details.
- `whence`: Position in file. See `fseek` for details.

int **esp_appttrace_ftell** (*esp_appttrace_dest_t* *dest*, void **stream*)

Get current position indicator for file on host. This function has the same semantic as 'ftell' except for the first argument.

Return Current position in file. See `ftell` for details.

Parameters

- `dest`: Indicates HW interface to use.
- `stream`: File handle returned by `esp_appttrace_fopen`.

int **esp_appttrace_fstop** (*esp_appttrace_dest_t* *dest*)

Indicates to the host that all file operations are completed. This function should be called after all file operations are finished and indicate to the host that it can perform cleanup operations (close open files etc.).

Return ESP_OK on success, otherwise see `esp_err_t`

Parameters

- `dest`: Indicates HW interface to use.

void **esp_gcov_dump** (void)

Triggers gcov info dump. This function waits for the host to connect to target before dumping data.

Enumerations

enum **esp_appttrace_dest_t**

Application trace data destinations bits.

Values:

ESP_APPTRACE_DEST_TRAX = 0x1
JTAG destination.

ESP_APPTRACE_DEST_UART0 = 0x2
UART destination.

Header File

- [app_trace/include/esp_sysview_trace.h](#)

Functions

static *esp_err_t* **esp_sysview_flush** (uint32_t *tmo*)

Flushes remaining data in SystemView trace buffer to host.

Return ESP_OK.

Parameters

- `tmo`: Timeout for operation (in us). Use `ESP_APPTRACE_TMO_INFINITE` to wait indefinitely.

int **esp_sysview_vprintf** (**const** char **format*, va_list *args*)

vprintf-like function to sent log messages to the host.

Return Number of bytes written.

Parameters

- `format`: Address of format string.
- `args`: List of arguments.

esp_err_t **esp_sysview_heap_trace_start** (uint32_t *tmo*)

Starts SystemView heap tracing.

Return ESP_OK on success, ESP_ERR_TIMEOUT if operation has been timed out.

Parameters

- `tmo`: Timeout (in us) to wait for the host to be connected. Use -1 to wait forever.

`esp_err_t esp_sysview_heap_trace_stop` (void)

Stops SystemView heap tracing.

Return ESP_OK.

void `esp_sysview_heap_trace_alloc` (void **addr*, uint32_t *size*, const void **callers*)

Sends heap allocation event to the host.

Parameters

- `addr`: Address of allocated block.
- `size`: Size of allocated block.
- `callers`: Pointer to array with callstack addresses. Array size must be CONFIG_HEAP_TRACING_STACK_DEPTH.

void `esp_sysview_heap_trace_free` (void **addr*, const void **callers*)

Sends heap de-allocation event to the host.

Parameters

- `addr`: Address of de-allocated block.
- `callers`: Pointer to array with callstack addresses. Array size must be CONFIG_HEAP_TRACING_STACK_DEPTH.

2.7.3 eFuse Manager

Introduction

The eFuse Manager library is designed to structure access to eFuse bits and make using these easy. This library operates eFuse bits by a structure name which is assigned in eFuse table. This sections introduces some concepts used by eFuse Manager.

Hardware description

The ESP32 has a number of eFuses which can store system and user parameters. Each eFuse is a one-bit field which can be programmed to 1 after which it cannot be reverted back to 0. Some of system parameters are using these eFuse bits directly by hardware modules and have special place (for example EFUSE_BLK0). For more details see [ESP32 Technical Reference Manual](#) in part 20 eFuse controller. Some eFuse bits are available for user applications.

ESP32 has 4 eFuse blocks each of the size of 256 bits (not all bits are available):

- EFUSE_BLK0 is used entirely for system purposes;
- EFUSE_BLK1 is used for flash encrypt key. If not using that Flash Encryption feature, they can be used for another purpose;
- EFUSE_BLK2 is used for security boot key. If not using that Secure Boot feature, they can be used for another purpose;
- EFUSE_BLK3 can be partially reserved for the custom MAC address, or used entirely for user application. Note that some bits are already used in IDF.

Each block is divided into 8 32-bits registers.

eFuse Manager component

The component has API functions for reading and writing fields. Access to the fields is carried out through the structures that describe the location of the eFuse bits in the blocks. The component provides the ability to form fields of any length and from any number of individual bits. The description of the fields is made in a CSV file in a table form. To generate from a tabular form (CSV file) in the C-source uses the tool `efuse_table_gen.py`. The tool checks the CSV file for uniqueness of field names and bit intersection, in case of using a `custom` file from the user's project directory, the utility will check with the `common` CSV file.

CSV files:

- common (*esp_efuse_table.csv*) - contains eFuse fields which are used inside the IDF. C-source generation should be done manually when changing this file (run command `idf.py efuse_common_table`). Note that changes in this file can lead to incorrect operation.
- custom - (optional and can be enabled by `CONFIG_EFUSE_CUSTOM_TABLE`) contains eFuse fields that are used by the user in their application. C-source generation should be done manually when changing this file and running `idf.py efuse_custom_table`.

Description CSV file

The CSV file contains a description of the eFuse fields. In the simple case, one field has one line of description. Table header:

```
# field_name, efuse_block(EFUSE_BLK0..EFUSE_BLK3), bit_start(0..255), bit_
↪count(1..256), comment
```

Individual params in CSV file the following meanings:

field_name Name of field. The prefix `ESP_EFUSE_` will be added to the name, and this field name will be available in the code. This name will be used to access the fields. The name must be unique for all fields. If the line has an empty name, then this line is combined with the previous field. This allows you to set an arbitrary order of bits in the field, and expand the field as well (see `MAC_FACTORY` field in the common table).

efuse_block Block number. It determines where the eFuse bits will be placed for this field. Available `EFUSE_BLK0..EFUSE_BLK3`.

bit_start Start bit number (0..255). The `bit_start` field can be omitted. In this case, it will be set to `bit_start + bit_count` from the previous record, if it has the same `efuse_block`. Otherwise (if `efuse_block` is different, or this is the first entry), an error will be generated.

bit_count The number of bits to use in this field (1..-). This parameter can not be omitted. This field also may be `MAX_BLK_LEN` in this case, the field length will have the maximum block length, taking into account the coding scheme (applicable for `ESP_EFUSE_SECURE_BOOT_KEY` and `ESP_EFUSE_ENCRYPT_FLASH_KEY` fields). The value `MAX_BLK_LEN` depends on `CONFIG_EFUSE_MAX_BLK_LEN`, will be replaced with “None” - 256, “3/4” - 192, “REPEAT” - 128.

comment This param is using for comment field, it also move to C-header file. The comment field can be omitted.

If a non-sequential bit order is required to describe a field, then the field description in the following lines should be continued without specifying a name, this will indicate that it belongs to one field. For example two fields `MAC_FACTORY` and `MAC_FACTORY_CRC`:

```
# Factory MAC address #
#####
MAC_FACTORY,          EFUSE_BLK0,    72,    8,    Factory MAC addr [0]
,                    EFUSE_BLK0,    64,    8,    Factory MAC addr [1]
,                    EFUSE_BLK0,    56,    8,    Factory MAC addr [2]
,                    EFUSE_BLK0,    48,    8,    Factory MAC addr [3]
,                    EFUSE_BLK0,    40,    8,    Factory MAC addr [4]
,                    EFUSE_BLK0,    32,    8,    Factory MAC addr [5]
MAC_FACTORY_CRC,     EFUSE_BLK0,    80,    8,    CRC8 for factory MAC address
```

This field will available in code as `ESP_EFUSE_MAC_FACTORY` and `ESP_EFUSE_MAC_FACTORY_CRC`.

efuse_table_gen.py tool

The tool is designed to generate C-source files from CSV file and validate fields. First of all, the check is carried out on the uniqueness of the names and overlaps of the field bits. If an additional *custom* file is used, it will be checked with the existing *common* file (*esp_efuse_table.csv*). In case of errors, a message will be displayed and the string that caused the error. C-source files contain structures of type `esp_efuse_desc_t`.

To generate a *common* files, use the following command `idf.py efuse_common_table` or:

```
cd $IDF_PATH/components/efuse/  
./efuse_table_gen.py esp32/esp_efuse_table.csv
```

After generation in the folder *esp32* create:

- *esp_efuse_table.c* file.
- In *include* folder *esp_efuse_table.c* file.

To generate a *custom* files, use the following command `idf.py efuse_custom_table` or:

```
cd $IDF_PATH/components/efuse/  
./efuse_table_gen.py esp32/esp_efuse_table.csv PROJECT_PATH/main/esp_efuse_custom_  
→table.csv
```

After generation in the folder `PROJECT_PATH/main` create:

- *esp_efuse_custom_table.c* file.
- In *include* folder *esp_efuse_custom_table.c* file.

To use the generated fields, you need to include two files:

```
#include "esp_efuse.h"  
#include "esp_efuse_table.h" or "esp_efuse_custom_table.h"
```

Support coding scheme

eFuse have three coding schemes:

- None (value 0).
- 3/4 (value 1).
- Repeat (value 2).

The coding scheme affects only EFUSE_BLK1, EFUSE_BLK2 and EFUSE_BLK3 blocks. EUSE_BLK0 block always has a coding scheme None. Coding changes the number of bits that can be written into a block, the block length is constant 256, some of these bits are used for encoding and are not used.

When using a coding scheme, the length of the payload that can be written is limited (for more details 20.3.1.3 System Parameter `coding_scheme`):

- None 256 bits.
- 3/4 192 bits.
- Repeat 128 bits.

You can find out the coding scheme of your chip:

- run a `espefuse.py -p COM4 summary` command.
- from `esptool` utility logs (during flashing).
- calling the function in the code `esp_efuse_get_coding_scheme()` for the EFUSE_BLK3 block.

eFuse tables must always comply with the coding scheme in the chip. There is an `EFUSE_CODE_SCHEME_SELECTOR` option to select the coding type for tables in a Kconfig. When generating source files, if your tables do not follow the coding scheme, an error message will be displayed. Adjust the length or offset fields. If your program was compiled with None encoding and 3/4 is used in the chip, then the `ESP_ERR_CODING` error may occur when calling the eFuse API (the field is outside the block boundaries). If the field matches the new block boundaries, then the API will work without errors.

Also, 3/4 coding scheme imposes restrictions on writing bits belonging to one coding unit. The whole block with a length of 256 bits is divided into 4 coding units, and in each coding unit there are 6 bytes of useful data and 2 service bytes. These 2 service bytes contain the checksum of the previous 6 data bytes.

It turns out that only one field can be written into one coding unit. Repeated rewriting in one coding unit is prohibited. But if the record was made in advance or through a `esp_efuse_write_block()` function, then reading the fields belonging to one coding unit is possible.

In case 3/4 coding scheme, the writing process is divided into the coding units and we can not use the usual mode of writing some fields. We can prepare all the data for writing and burn it in one time. You can also use this mode for None coding scheme but it is not necessary. It is important for 3/4 coding scheme. To write some fields in one time need to use the batch writing mode. Firstly set this mode through `esp_efuse_batch_write_begin()` function then write some fields as usual use the `esp_efuse_write_...` functions. At the end to burn they, need to call the `esp_efuse_batch_write_commit()` function. It burns prepared data to the efuse blocks and disable the batch recording mode. The batch writing mode blocks `esp_efuse_read_...` operations.

After changing the coding scheme, run `efuse_common_table` and `efuse_custom_table` commands to check the tables of the new coding scheme.

eFuse API

Access to the fields is via a pointer to the description structure. API functions have some basic operation:

- `esp_efuse_read_field_blob()` - returns an array of read eFuse bits.
- `esp_efuse_read_field_cnt()` - returns the number of bits programmed as “1” .
- `esp_efuse_write_field_blob()` - writes an array.
- `esp_efuse_write_field_cnt()` - writes a required count of bits as “1” .
- `esp_efuse_get_field_size()` - returns the number of bits by the field name.
- `esp_efuse_read_reg()` - returns value of eFuse register.
- `esp_efuse_write_reg()` - writes value to eFuse register.
- `esp_efuse_get_coding_scheme()` - returns eFuse coding scheme for blocks.
- `esp_efuse_read_block()` - reads key to eFuse block starting at the offset and the required size.
- `esp_efuse_write_block()` - writes key to eFuse block starting at the offset and the required size.
- `esp_efuse_batch_write_begin()` - set the batch mode of writing fields.
- `esp_efuse_batch_write_commit()` - writes all prepared data for batch writing mode and reset the batch writing mode.
- `esp_efuse_batch_write_cancel()` - reset the batch writing mode and prepared data.

For frequently used fields, special functions are made, like this `esp_efuse_get_chip_ver()`, `esp_efuse_get_pkg_ver()`.

How add a new field

1. Find a free bits for field. Show `esp_efuse_table.csv` file or run `idf.py show_efuse_table` or the next command:

```
$ ./efuse_table_gen.py esp32/esp_efuse_table.csv --info
eFuse coding scheme: NONE
#       field_name           efuse_block    bit_start      bit_count
1       WR_DIS_FLASH_CRYPT_CNT EFUSE_BLK0     2              1
2       WR_DIS_BLK1          EFUSE_BLK0     7              1
3       WR_DIS_BLK2          EFUSE_BLK0     8              1
4       WR_DIS_BLK3          EFUSE_BLK0     9              1
5       RD_DIS_BLK1          EFUSE_BLK0     16             1
6       RD_DIS_BLK2          EFUSE_BLK0     17             1
7       RD_DIS_BLK3          EFUSE_BLK0     18             1
8       FLASH_CRYPT_CNT      EFUSE_BLK0     20             7
9       MAC_FACTORY          EFUSE_BLK0     32             8
10      MAC_FACTORY          EFUSE_BLK0     40             8
11      MAC_FACTORY          EFUSE_BLK0     48             8
12      MAC_FACTORY          EFUSE_BLK0     56             8
13      MAC_FACTORY          EFUSE_BLK0     64             8
14      MAC_FACTORY          EFUSE_BLK0     72             8
15      MAC_FACTORY_CRC      EFUSE_BLK0     80             8
16      CHIP_VER_DIS_APP_CPU  EFUSE_BLK0     96             1
17      CHIP_VER_DIS_BT       EFUSE_BLK0     97             1
```

(continues on next page)

(continued from previous page)

| | | | | |
|----|------------------------|------------|-----|-----|
| 18 | CHIP_VER_PKG | EFUSE_BLK0 | 105 | 3 |
| 19 | CHIP_CPU_FREQ_LOW | EFUSE_BLK0 | 108 | 1 |
| 20 | CHIP_CPU_FREQ_RATED | EFUSE_BLK0 | 109 | 1 |
| 21 | CHIP_VER_REV1 | EFUSE_BLK0 | 111 | 1 |
| 22 | ADC_VREF_AND_SDIO_DREF | EFUSE_BLK0 | 136 | 6 |
| 23 | XPD_SDIO_REG | EFUSE_BLK0 | 142 | 1 |
| 24 | SDIO_TIEH | EFUSE_BLK0 | 143 | 1 |
| 25 | SDIO_FORCE | EFUSE_BLK0 | 144 | 1 |
| 26 | ENCRYPT_CONFIG | EFUSE_BLK0 | 188 | 4 |
| 27 | CONSOLE_DEBUG_DISABLE | EFUSE_BLK0 | 194 | 1 |
| 28 | ABS_DONE_0 | EFUSE_BLK0 | 196 | 1 |
| 29 | DISABLE_JTAG | EFUSE_BLK0 | 198 | 1 |
| 30 | DISABLE_DL_ENCRYPT | EFUSE_BLK0 | 199 | 1 |
| 31 | DISABLE_DL_DECRYPT | EFUSE_BLK0 | 200 | 1 |
| 32 | DISABLE_DL_CACHE | EFUSE_BLK0 | 201 | 1 |
| 33 | ENCRYPT_FLASH_KEY | EFUSE_BLK1 | 0 | 256 |
| 34 | SECURE_BOOT_KEY | EFUSE_BLK2 | 0 | 256 |
| 35 | MAC_CUSTOM_CRC | EFUSE_BLK3 | 0 | 8 |
| 36 | MAC_CUSTOM | EFUSE_BLK3 | 8 | 48 |
| 37 | ADC1_TP_LOW | EFUSE_BLK3 | 96 | 7 |
| 38 | ADC1_TP_HIGH | EFUSE_BLK3 | 103 | 9 |
| 39 | ADC2_TP_LOW | EFUSE_BLK3 | 112 | 7 |
| 40 | ADC2_TP_HIGH | EFUSE_BLK3 | 119 | 9 |
| 41 | SECURE_VERSION | EFUSE_BLK3 | 128 | 32 |
| 42 | MAC_CUSTOM_VER | EFUSE_BLK3 | 184 | 8 |

Used bits in eFuse table:

EFUSE_BLK0

[2 2] [7 9] [16 18] [20 27] [32 87] [96 97] [105 109] [111 111] [136 144] [188_↵
↵191] [194 194] [196 196] [198 201]

EFUSE_BLK1

[0 255]

EFUSE_BLK2

[0 255]

EFUSE_BLK3

[0 55] [96 159] [184 191]

Note: Not printed ranges are free for using. (bits in EFUSE_BLK0 are reserved for_↵
↵Espressif)

Parsing eFuse CSV input file \$IDF_PATH/components/efuse/esp32/esp_efuse_table.csv .
↵..

Verifying eFuse table...

The number of bits not included in square brackets is free (bits in EFUSE_BLK0 are reserved for Espressif). All fields are checked for overlapping.

2. Fill a line for field: field_name, efuse_block, bit_start, bit_count, comment.
3. Run a show_efuse_table command to check eFuse table. To generate source files run efuse_common_table or efuse_custom_table command.

Debug eFuse & Unit tests

Virtual eFuses The Kconfig option CONFIG_EFUSE_VIRTUAL will virtualize eFuse values inside the eFuse Manager, so writes are emulated and no eFuse values are permanently changed. This can be useful for debugging app and unit tests.

| | | | |
|---|---|--------|-----|
| Identity fuses: | | | |
| MAC | MAC Address | | |
| = 84:0d:8e:18:8e:44 | (CRC ad OK) R/W | | |
| CHIP_VER_REV1 | Silicon Revision 1 | = 1 | R/W |
| ↔ (0x1) | | | |
| CHIP_VERSION | Reserved for future chip versions | = 2 | R/W |
| ↔ (0x2) | | | |
| CHIP_PACKAGE | Chip package identifier | = 0 | R/W |
| ↔ (0x0) | | | |
| Calibration fuses: | | | |
| BLK3_PART_RESERVE | BLOCK3 partially served for ADC calibration data | = 1 | R/W |
| ↔ (0x1) | | | |
| ADC_VREF | Voltage reference calibration | = 1114 | R/ |
| ↔W (0x2) | | | |
| ADC1_TP_LOW | ADC1 150mV reading | = 346 | R/W |
| ↔ (0x11) | | | |
| ADC1_TP_HIGH | ADC1 850mV reading | = 3285 | R/ |
| ↔W (0x5) | | | |
| ADC2_TP_LOW | ADC2 150mV reading | = 449 | R/W |
| ↔ (0x7) | | | |
| ADC2_TP_HIGH | ADC2 850mV reading | = 3362 | R/ |
| ↔W (0x1f5) | | | |
| Flash voltage (VDD_SDIO) determined by GPIO12 on reset (High for 1.8V, Low/NC for ↔3.3V). | | | |

To get a dump for all eFuse registers.

```
espefuse.py -p COM4 dump

$ espefuse.py -p COM4 dump
espefuse.py v2.3.1
Connecting.....__
EFUSE block 0:
00000000 c403bb68 0082240a 00000000 00000035 00000000 00000000
EFUSE block 1:
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
EFUSE block 2:
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
EFUSE block 3:
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Header File

- [efuse/include/esp_efuse.h](#)

Functions

`esp_err_t esp_efuse_read_field_blob(const esp_efuse_desc_t *field[], void *dst, size_t dst_size_bits)`

Reads bits from EFUSE field and writes it into an array.

The number of read bits will be limited to the minimum value from the description of the bits in “field” structure or “dst_size_bits” required size. Use “esp_efuse_get_field_size()” function to determine the length of the field.

Return

- ESP_OK: The operation was successfully completed.
- ESP_ERR_INVALID_ARG: Error in the passed arguments.

Parameters

- [in] `field`: A pointer to the structure describing the fields of efuse.
- [out] `dst`: A pointer to array that will contain the result of reading.
- [in] `dst_size_bits`: The number of bits required to read. If the requested number of bits is greater than the field, the number will be limited to the field size.

bool **esp_efuse_read_field_bit** (const *esp_efuse_desc_t* *`field`[])

Read a single bit eFuse field as a boolean value.

Note The value must exist and must be a single bit wide. If there is any possibility of an error in the provided arguments, call `esp_efuse_read_field_blob()` and check the returned value instead.

Note If assertions are enabled and the parameter is invalid, execution will abort

Return

- true: The field parameter is valid and the bit is set.
- false: The bit is not set, or the parameter is invalid and assertions are disabled.

Parameters

- [in] `field`: A pointer to the structure describing the fields of efuse.

esp_err_t **esp_efuse_read_field_cnt** (const *esp_efuse_desc_t* *`field`[], *size_t* *`out_cnt`)

Reads bits from EFUSE field and returns number of bits programmed as “1” .

If the bits are set not sequentially, they will still be counted.

Return

- ESP_OK: The operation was successfully completed.
- ESP_ERR_INVALID_ARG: Error in the passed arguments.

Parameters

- [in] `field`: A pointer to the structure describing the fields of efuse.
- [out] `out_cnt`: A pointer that will contain the number of programmed as “1” bits.

esp_err_t **esp_efuse_write_field_blob** (const *esp_efuse_desc_t* *`field`[], const void *`src`, *size_t* `src_size_bits`)

Writes array to EFUSE field.

The number of write bits will be limited to the minimum value from the description of the bits in “field” structure or “src_size_bits” required size. Use “`esp_efuse_get_field_size()`” function to determine the length of the field. After the function is completed, the writing registers are cleared.

Return

- ESP_OK: The operation was successfully completed.
- ESP_ERR_INVALID_ARG: Error in the passed arguments.
- ESP_ERR_EFUSE_REPEATED_PROG: Error repeated programming of programmed bits is strictly forbidden.
- ESP_ERR_CODING: Error range of data does not match the coding scheme.

Parameters

- [in] `field`: A pointer to the structure describing the fields of efuse.
- [in] `src`: A pointer to array that contains the data for writing.
- [in] `src_size_bits`: The number of bits required to write.

esp_err_t **esp_efuse_write_field_cnt** (const *esp_efuse_desc_t* *`field`[], *size_t* `cnt`)

Writes a required count of bits as “1” to EFUSE field.

If there are no free bits in the field to set the required number of bits to “1” , ESP_ERR_EFUSE_CNT_IS_FULL error is returned, the field will not be partially recorded. After the function is completed, the writing registers are cleared.

Return

- ESP_OK: The operation was successfully completed.
- ESP_ERR_INVALID_ARG: Error in the passed arguments.
- ESP_ERR_EFUSE_CNT_IS_FULL: Not all requested cnt bits is set.

Parameters

- [in] `field`: A pointer to the structure describing the fields of efuse.
- [in] `cnt`: Required number of programmed as “1” bits.

esp_err_t **esp_efuse_write_field_bit** (const *esp_efuse_desc_t* *field[])

Write a single bit eFuse field to 1.

For use with eFuse fields that are a single bit. This function will write the bit to value 1 if it is not already set, or does nothing if the bit is already set.

This is equivalent to calling `esp_efuse_write_field_cnt()` with the `cnt` parameter equal to 1, except that it will return `ESP_OK` if the field is already set to 1.

Return

- `ESP_OK`: The operation was successfully completed, or the bit was already set to value 1.
- `ESP_ERR_INVALID_ARG`: Error in the passed arguments, including if the eFuse field is not 1 bit wide.

Parameters

- [in] `field`: Pointer to the structure describing the eFuse field.

esp_err_t **esp_efuse_set_write_protect** (*esp_efuse_block_t* blk)

Sets a write protection for the whole block.

After that, it is impossible to write to this block. The write protection does not apply to block 0.

Return

- `ESP_OK`: The operation was successfully completed.
- `ESP_ERR_INVALID_ARG`: Error in the passed arguments.
- `ESP_ERR_EFUSE_CNT_IS_FULL`: Not all requested `cnt` bits is set.
- `ESP_ERR_NOT_SUPPORTED`: The block does not support this command.

Parameters

- [in] `blk`: Block number of eFuse. (`EFUSE_BLK1`, `EFUSE_BLK2` and `EFUSE_BLK3`)

esp_err_t **esp_efuse_set_read_protect** (*esp_efuse_block_t* blk)

Sets a read protection for the whole block.

After that, it is impossible to read from this block. The read protection does not apply to block 0.

Return

- `ESP_OK`: The operation was successfully completed.
- `ESP_ERR_INVALID_ARG`: Error in the passed arguments.
- `ESP_ERR_EFUSE_CNT_IS_FULL`: Not all requested `cnt` bits is set.
- `ESP_ERR_NOT_SUPPORTED`: The block does not support this command.

Parameters

- [in] `blk`: Block number of eFuse. (`EFUSE_BLK1`, `EFUSE_BLK2` and `EFUSE_BLK3`)

int **esp_efuse_get_field_size** (const *esp_efuse_desc_t* *field[])

Returns the number of bits used by field.

Return Returns the number of bits used by field.

Parameters

- [in] `field`: A pointer to the structure describing the fields of eFuse.

uint32_t **esp_efuse_read_reg** (*esp_efuse_block_t* blk, unsigned int *num_reg*)

Returns value of eFuse register.

This is a thread-safe implementation. Example: `EFUSE_BLK2_RDATA3_REG` where (`blk=2`, `num_reg=3`)

Return Value of register

Parameters

- [in] `blk`: Block number of eFuse.
- [in] `num_reg`: The register number in the block.

esp_err_t **esp_efuse_write_reg** (*esp_efuse_block_t* blk, unsigned int *num_reg*, uint32_t *val*)

Write value to eFuse register.

Apply a coding scheme if necessary. This is a thread-safe implementation. Example: `EFUSE_BLK3_WDATA0_REG` where (`blk=3`, `num_reg=0`)

Return

- `ESP_OK`: The operation was successfully completed.

- `ESP_ERR_EFUSE_REPEATED_PROG`: Error repeated programming of programmed bits is strictly forbidden.

Parameters

- `[in] blk`: Block number of eFuse.
- `[in] num_reg`: The register number in the block.
- `[in] val`: Value to write.

`esp_efuse_coding_scheme_t esp_efuse_get_coding_scheme` (`esp_efuse_block_t blk`)

Return efuse coding scheme for blocks.

Note: The coding scheme is applicable only to 1, 2 and 3 blocks. For 0 block, the coding scheme is always NONE.

Return Return efuse coding scheme for blocks

Parameters

- `[in] blk`: Block number of eFuse.

`esp_err_t esp_efuse_read_block` (`esp_efuse_block_t blk`, `void *dst_key`, `size_t offset_in_bits`, `size_t size_bits`)

Read key to efuse block starting at the offset and the required size.

Return

- `ESP_OK`: The operation was successfully completed.
- `ESP_ERR_INVALID_ARG`: Error in the passed arguments.
- `ESP_ERR_CODING`: Error range of data does not match the coding scheme.

Parameters

- `[in] blk`: Block number of eFuse.
- `[in] dst_key`: A pointer to array that will contain the result of reading.
- `[in] offset_in_bits`: Start bit in block.
- `[in] size_bits`: The number of bits required to read.

`esp_err_t esp_efuse_write_block` (`esp_efuse_block_t blk`, `const void *src_key`, `size_t offset_in_bits`, `size_t size_bits`)

Write key to efuse block starting at the offset and the required size.

Return

- `ESP_OK`: The operation was successfully completed.
- `ESP_ERR_INVALID_ARG`: Error in the passed arguments.
- `ESP_ERR_CODING`: Error range of data does not match the coding scheme.
- `ESP_ERR_EFUSE_REPEATED_PROG`: Error repeated programming of programmed bits

Parameters

- `[in] blk`: Block number of eFuse.
- `[in] src_key`: A pointer to array that contains the key for writing.
- `[in] offset_in_bits`: Start bit in block.
- `[in] size_bits`: The number of bits required to write.

`uint8_t esp_efuse_get_chip_ver` (void)

Returns chip version from efuse.

Return chip version

`uint32_t esp_efuse_get_pkg_ver` (void)

Returns chip package from efuse.

Return chip package

void `esp_efuse_burn_new_values` (void)

void `esp_efuse_reset` (void)

void `esp_efuse_disable_basic_rom_console` (void)

`esp_err_t esp_efuse_disable_rom_download_mode` (void)

void `esp_efuse_write_random_key` (`uint32_t blk_wdata0_reg`)

`uint32_t esp_efuse_read_secure_version` (void)

```
bool esp_efuse_check_secure_version (uint32_t secure_version)
esp_err_t esp_efuse_update_secure_version (uint32_t secure_version)
void esp_efuse_init (uint32_t offset, uint32_t size)
esp_err_t esp_efuse_batch_write_begin (void)
esp_err_t esp_efuse_batch_write_cancel (void)
esp_err_t esp_efuse_batch_write_commit (void)
static bool soc_has_cache_lock_bug (void)
```

Structures

```
struct esp_efuse_desc_t
    Structure eFuse field.
```

Public Members

```
esp_efuse_block_t efuse_block : 8
    Block of eFuse

uint8_t bit_start
    Start bit [0..255]

uint16_t bit_count
    Length of bit field [1..-]
```

Macros

```
ESP_ERR_EFUSE
    Base error code for efuse api.

ESP_OK_EFUSE_CNT
    OK the required number of bits is set.

ESP_ERR_EFUSE_CNT_IS_FULL
    Error field is full.

ESP_ERR_EFUSE_REPEATED_PROG
    Error repeated programming of programmed bits is strictly forbidden.

ESP_ERR_CODING
    Error while a encoding operation.
```

2.7.4 Error Codes and Helper Functions

This section lists definitions of common ESP-IDF error codes and several helper functions related to error handling.

For general information about error codes in ESP-IDF, see [Error Handling](#).

For the full list of error codes defined in ESP-IDF, see [Error Code Reference](#).

API Reference

Header File

- [esp_common/include/esp_err.h](#)

Functions

const char ***esp_err_to_name** (*esp_err_t* code)

Returns string for esp_err_t error codes.

This function finds the error code in a pre-generated lookup-table and returns its string representation.

The function is generated by the Python script tools/gen_esp_err_to_name.py which should be run each time an esp_err_t error is modified, created or removed from the IDF project.

Return string error message

Parameters

- code: esp_err_t error code

const char ***esp_err_to_name_r** (*esp_err_t* code, char *buf, size_t buflen)

Returns string for esp_err_t and system error codes.

This function finds the error code in a pre-generated lookup-table of esp_err_t errors and returns its string representation. If the error code is not found then it is attempted to be found among system errors.

The function is generated by the Python script tools/gen_esp_err_to_name.py which should be run each time an esp_err_t error is modified, created or removed from the IDF project.

Return buf containing the string error message

Parameters

- code: esp_err_t error code
- [out] buf: buffer where the error message should be written
- buflen: Size of buffer buf. At most buflen bytes are written into the buf buffer (including the terminating null byte).

Macros

ESP_OK

esp_err_t value indicating success (no error)

ESP_FAIL

Generic esp_err_t code indicating failure

ESP_ERR_NO_MEM

Out of memory

ESP_ERR_INVALID_ARG

Invalid argument

ESP_ERR_INVALID_STATE

Invalid state

ESP_ERR_INVALID_SIZE

Invalid size

ESP_ERR_NOT_FOUND

Requested resource not found

ESP_ERR_NOT_SUPPORTED

Operation or feature not supported

ESP_ERR_TIMEOUT

Operation timed out

ESP_ERR_INVALID_RESPONSE

Received response was invalid

ESP_ERR_INVALID_CRC

CRC or checksum was invalid

ESP_ERR_INVALID_VERSION

Version was invalid

ESP_ERR_INVALID_MAC

MAC address was invalid

ESP_ERR_WIFI_BASE

Starting number of WiFi error codes

ESP_ERR_MESH_BASE

Starting number of MESH error codes

ESP_ERR_FLASH_BASE

Starting number of flash error codes

ESP_ERROR_CHECK (x)

Macro which can be used to check the error code, and terminate the program in case the code is not ESP_OK. Prints the error code, error location, and the failed statement to serial output.

Disabled if assertions are disabled.

ESP_ERROR_CHECK_WITHOUT_ABORT (x)

Macro which can be used to check the error code. Prints the error code, error location, and the failed statement to serial output. In comparison with ESP_ERROR_CHECK(), this prints the same error message but isn't terminating the program.

Type Definitions

```
typedef int32_t esp_err_t
```

2.7.5 ESP HTTPS OTA

Overview

esp_https_ota provides simplified APIs to perform firmware upgrades over HTTPS. It's an abstraction layer over existing OTA APIs.

Application Example

```
esp_err_t do_firmware_upgrade()
{
    esp_http_client_config_t config = {
        .url = CONFIG_FIRMWARE_UPGRADE_URL,
        .cert_pem = (char *)server_cert_pem_start,
    };
    esp_err_t ret = esp_https_ota(&config);
    if (ret == ESP_OK) {
        esp_restart();
    } else {
        return ESP_FAIL;
    }
    return ESP_OK;
}
```

Signature Verification

For additional security, signature of OTA firmware images can be verified. For that, refer [Secure OTA Updates Without Secure boot](#)

API Reference**Header File**

- [esp_https_ota/include/esp_https_ota.h](#)

Functions

esp_err_t esp_https_ota (**const** *esp_http_client_config_t* **config*)

HTTPS OTA Firmware upgrade.

This function allocates HTTPS OTA Firmware upgrade context, establishes HTTPS connection, reads image data from HTTP stream and writes it to OTA partition and finishes HTTPS OTA Firmware upgrade operation. This API supports URL redirection, but if CA cert of URLs differ then it should be appended to `cert_pem` member of `config`.

Note This API handles the entire OTA operation, so if this API is being used then no other APIs from `esp_https_ota` component should be called. If more information and control is needed during the HTTPS OTA process, then one can use `esp_https_ota_begin` and subsequent APIs. If this API returns successfully, `esp_restart()` must be called to boot from the new firmware image.

Return

- ESP_OK: OTA data updated, next reboot will use specified partition.
- ESP_FAIL: For generic failure.
- ESP_ERR_INVALID_ARG: Invalid argument
- ESP_ERR_OTA_VALIDATE_FAILED: Invalid app image
- ESP_ERR_NO_MEM: Cannot allocate memory for OTA operation.
- ESP_ERR_FLASH_OP_TIMEOUT or ESP_ERR_FLASH_OP_FAIL: Flash write failed.
- For other return codes, refer OTA documentation in `esp-idf`'s `app_update` component.

Parameters

- [in] `config`: pointer to *esp_http_client_config_t* structure.

esp_err_t esp_https_ota_begin (*esp_https_ota_config_t* **ota_config*, *esp_https_ota_handle_t* **handle*)

Start HTTPS OTA Firmware upgrade.

This function initializes ESP HTTPS OTA context and establishes HTTPS connection. This function must be invoked first. If this function returns successfully, then `esp_https_ota_perform` should be called to continue with the OTA process and there should be a call to `esp_https_ota_finish` on completion of OTA operation or on failure in subsequent operations. This API supports URL redirection, but if CA cert of URLs differ then it should be appended to `cert_pem` member of `http_config`, which is a part of `ota_config`. In case of error, this API explicitly sets `handle` to NULL.

Note This API is blocking, so setting `is_async` member of `http_config` structure will result in an error.

Return

- ESP_OK: HTTPS OTA Firmware upgrade context initialised and HTTPS connection established
- ESP_FAIL: For generic failure.
- ESP_ERR_INVALID_ARG: Invalid argument (missing/incorrect config, certificate, etc.)
- For other return codes, refer documentation in `app_update` component and `esp_http_client` component in `esp-idf`.

Parameters

- [in] `ota_config`: pointer to *esp_https_ota_config_t* structure
- [out] `handle`: pointer to an allocated data of type `esp_https_ota_handle_t` which will be initialised in this function

esp_err_t esp_https_ota_perform (*esp_https_ota_handle_t* *https_ota_handle*)

Read image data from HTTP stream and write it to OTA partition.

This function reads image data from HTTP stream and writes it to OTA partition. This function must be called only if `esp_https_ota_begin()` returns successfully. This function must be called in a loop since it returns after every HTTP read operation thus giving you the flexibility to stop OTA operation midway.

Return

- ESP_ERR_HTTPS_OTA_IN_PROGRESS: OTA update is in progress, call this API again to continue.
- ESP_OK: OTA update was successful
- ESP_FAIL: OTA update failed
- ESP_ERR_INVALID_ARG: Invalid argument
- ESP_ERR_OTA_VALIDATE_FAILED: Invalid app image
- ESP_ERR_NO_MEM: Cannot allocate memory for OTA operation.
- ESP_ERR_FLASH_OP_TIMEOUT or ESP_ERR_FLASH_OP_FAIL: Flash write failed.

- For other return codes, refer OTA documentation in esp-idf's app_update component.

Parameters

- [in] `https_ota_handle`: pointer to `esp_https_ota_handle_t` structure

bool **esp_https_ota_is_complete_data_received** (*esp_https_ota_handle_t* `https_ota_handle`)

Checks if complete data was received or not.

Note This API can be called just before `esp_https_ota_finish()` to validate if the complete image was indeed received.

Return

- false
- true

Parameters

- [in] `https_ota_handle`: pointer to `esp_https_ota_handle_t` structure

esp_err_t **esp_https_ota_finish** (*esp_https_ota_handle_t* `https_ota_handle`)

Clean-up HTTPS OTA Firmware upgrade and close HTTPS connection.

This function closes the HTTP connection and frees the ESP HTTPS OTA context. This function switches the boot partition to the OTA partition containing the new firmware image.

Note If this API returns successfully, `esp_restart()` must be called to boot from the new firmware image

Return

- ESP_OK: Clean-up successful
- ESP_ERR_INVALID_STATE
- ESP_ERR_INVALID_ARG: Invalid argument
- ESP_ERR_OTA_VALIDATE_FAILED: Invalid app image

Parameters

- [in] `https_ota_handle`: pointer to `esp_https_ota_handle_t` structure

esp_err_t **esp_https_ota_get_img_desc** (*esp_https_ota_handle_t* `https_ota_handle`, *esp_app_desc_t* `*new_app_info`)

Reads app description from image header. The app description provides information like the “Firmware version” of the image.

Note This API can be called only after `esp_https_ota_begin()` and before `esp_https_ota_perform()`. Calling this API is not mandatory.

Return

- ESP_ERR_INVALID_ARG: Invalid arguments
- ESP_FAIL: Failed to read image descriptor
- ESP_OK: Successfully read image descriptor

Parameters

- [in] `https_ota_handle`: pointer to `esp_https_ota_handle_t` structure
- [out] `new_app_info`: pointer to an allocated `esp_app_desc_t` structure

int **esp_https_ota_get_image_len_read** (*esp_https_ota_handle_t* `https_ota_handle`)

Structures

struct `esp_https_ota_config_t`
ESP HTTPS OTA configuration.

Public Members

const *esp_http_client_config_t* `*http_config`
ESP HTTP client configuration

Macros

ESP_ERR_HTTPS_OTA_BASE

ESP_ERR_HTTPS_OTA_IN_PROGRESS

Type Definitions

```
typedef void *esp_https_ota_handle_t
```

2.7.6 ESP-pthread**Overview**

This module offers Espressif specific extensions to the pthread library that can be used to influence the behaviour of pthread.

- Stack size of the pthreads
- Priority of the created pthreads
- Inheriting this configuration across threads
- Thread name
- Core affinity / core pinning.

Example to tune the stack size of the pthread:

```
void * thread_func(void * p)
{
    printf("In thread_func\n");
    return NULL;
}

void app_main(void)
{
    pthread_t t1;

    esp_pthread_cfg_t cfg = esp_create_default_pthread_config();
    cfg.stack_size = (4 * 1024);
    esp_pthread_set_cfg(&cfg);

    pthread_create(&t1, NULL, thread_func);
}
```

The API can also be used for inheriting the settings across threads. For example:

```
void * my_thread2(void * p)
{
    /* This thread will inherit the stack size of 4K */
    printf("In my_thread2\n");

    return NULL;
}

void * my_thread1(void * p)
{
    printf("In my_thread1\n");
    pthread_t t2;
    pthread_create(&t2, NULL, my_thread2);

    return NULL;
}

void app_main(void)
{
    pthread_t t1;

    esp_pthread_cfg_t cfg = esp_create_default_pthread_config();
    cfg.stack_size = (4 * 1024);
    cfg.inherit_cfg = true;
    esp_pthread_set_cfg(&cfg);
}
```

(continues on next page)

```
pthread_create(&t1, NULL, my_thread1);  
}
```

API Reference

Header File

- [pthread/include/esp_thread.h](#)

Functions

esp_thread_cfg_t **esp_thread_get_default_config** (void)

Creates a default pthread configuration based on the values set via menuconfig.

Return A default configuration structure.

esp_err_t **esp_thread_set_cfg** (const *esp_thread_cfg_t* *cfg)

Configure parameters for creating pthread.

This API allows you to configure how the subsequent pthread_create() call will behave. This call can be used to setup configuration parameters like stack size, priority, configuration inheritance etc.

If the 'inherit' flag in the configuration structure is enabled, then the same configuration is also inherited in the thread subtree.

Note Passing non-NULL attributes to pthread_create() will override the stack_size parameter set using this API

Return

- ESP_OK if configuration was successfully set
- ESP_ERR_NO_MEM if out of memory
- ESP_ERR_INVALID_ARG if stack_size is less than PTHREAD_STACK_MIN

Parameters

- *cfg*: The pthread config parameters

esp_err_t **esp_thread_get_cfg** (*esp_thread_cfg_t* *p)

Get current pthread creation configuration.

This will retrieve the current configuration that will be used for creating threads.

Return

- ESP_OK if the configuration was available
- ESP_ERR_NOT_FOUND if a configuration wasn't previously set

Parameters

- *p*: Pointer to the pthread config structure that will be updated with the currently configured parameters

Structures

struct esp_thread_cfg_t

pthread configuration structure that influences pthread creation

Public Members

size_t **stack_size**

The stack size of the pthread.

size_t **prio**

The thread's priority.

bool **inherit_cfg**

Inherit this configuration further.

const char *thread_name

The thread name.

int pin_to_core

The core id to pin the thread to. Has the same value range as xCoreId argument of xTaskCreatePinnedToCore.

Macros

PTHREAD_STACK_MIN

2.7.7 Event Loop Library

Overview

The event loop library allows components to declare events to which other components can register handlers –code which will execute when those events occur. This allows loosely coupled components to attach desired behavior to changes in state of other components without application involvement. For instance, a high level connection handling library may subscribe to events produced by the wifi subsystem directly and act on those events. This also simplifies event processing by serializing and deferring code execution to another context.

Using esp_event APIs

There are two objects of concern for users of this library: events and event loops.

Events are occurrences of note. For example, for WiFi, a successful connection to the access point may be an event. Events are referenced using a two part identifier which are discussed more [here](#). Event loops are the vehicle by which events get posted by event sources and handled by event handler functions. These two appear prominently in the event loop library APIs.

Using this library roughly entails the following flow:

1. A user defines a function that should run when an event is posted to a loop. This function is referred to as the event handler. It should have the same signature as `esp_event_handler_t`.
2. An event loop is created using `esp_event_loop_create()`, which outputs a handle to the loop of type `esp_event_loop_handle_t`. Event loops created using this API are referred to as user event loops. There is, however, a special type of event loop called the default event loop which are discussed [here](#).
3. Components register event handlers to the loop using `esp_event_handler_register_with()`. Handlers can be registered with multiple loops, more on that [here](#).
4. Event sources post an event to the loop using `esp_event_post_to()`.
5. Components wanting to remove their handlers from being called can do so by unregistering from the loop using `esp_event_handler_unregister_with()`.
6. Event loops which are no longer needed can be deleted using `esp_event_loop_delete()`.

In code, the flow above may look like as follows:

```
// 1. Define the event handler
void run_on_event(void* handler_arg, esp_event_base_t base, int32_t id, void*
↳event_data)
{
    // Event handler logic
}

void app_main()
{
    // 2. A configuration structure of type esp_event_loop_args_t is needed to
↳specify the properties of the loop to be
    // created. A handle of type esp_event_loop_handle_t is obtained, which is
↳needed by the other APIs to reference the loop
    // to perform their operations on.
```

(continues on next page)

```

esp_event_loop_args_t loop_args = {
    .queue_size = ...,
    .task_name = ...
    .task_priority = ...,
    .task_stack_size = ...,
    .task_core_id = ...
};

esp_event_loop_handle_t loop_handle;

esp_event_loop_create(&loop_args, &loop_handle);

// 3. Register event handler defined in (1). MY_EVENT_BASE and MY_EVENT_ID
↳ specifies a hypothetical
// event that handler run_on_event should execute on when it gets posted to
↳ the loop.
esp_event_handler_register_with(loop_handle, MY_EVENT_BASE, MY_EVENT_ID, run_
↳ on_event, ...);

...

// 4. Post events to the loop. This queues the event on the event loop. At
↳ some point in time
// the event loop executes the event handler registered to the posted event,
↳ in this case run_on_event.
// For simplicity sake this example calls esp_event_post_to from app_main, but
↳ posting can be done from
// any other tasks (which is the more interesting use case).
esp_event_post_to(loop_handle, MY_EVENT_BASE, MY_EVENT_ID, ...);

...

// 5. Unregistering an unneeded handler
esp_event_handler_unregister_with(loop_handle, MY_EVENT_BASE, MY_EVENT_ID, run_
↳ on_event);

...

// 6. Deleting an unneeded event loop
esp_event_loop_delete(loop_handle);
}

```

Declaring and defining events

As mentioned previously, events consists of two-part identifiers: the event base and the event ID. The event base identifies an independent group of events; the event ID identifies the event within that group. Think of the event base and event ID as a person's last name and first name, respectively. A last name identifies a family, and the first name identifies a person within that family.

The event loop library provides macros to declare and define the event base easily.

Event base declaration:

```
ESP_EVENT_DECLARE_BASE(EVENT_BASE)
```

Event base definition:

```
ESP_EVENT_DEFINE_BASE(EVENT_BASE)
```

Note: In IDF, the base identifiers for system events are uppercase and are postfixed with `_EVENT`. For example,

the base for wifi events is declared and defined as `WIFI_EVENT`, the ethernet event base `ETHERNET_EVENT`, and so on. The purpose is to have event bases look like constants (although they are global variables considering the definitions of macros `ESP_EVENT_DECLARE_BASE` and `ESP_EVENT_DEFINE_BASE`).

For event ID's, declaring them as enumerations is recommended. Once again, for visibility, these are typically placed in public header files.

Event ID:

```
enum {
    EVENT_ID_1,
    EVENT_ID_2,
    EVENT_ID_3,
    ...
}
```

Default Event Loop

The default event loop is a special type of loop used for system events (WiFi events, for example). The handle for this loop is hidden from the user. The creation, deletion, handler registration/unregistration and posting of events is done through a variant of the APIs for user event loops. The table below enumerates those variants, and the user event loops equivalent.

| User Event Loops | Default Event Loops |
|--|--|
| <code>esp_event_loop_create()</code> | <code>esp_event_loop_create_default()</code> |
| <code>esp_event_loop_delete()</code> | <code>esp_event_loop_delete_default()</code> |
| <code>esp_event_handler_register_with()</code> | <code>esp_event_handler_register()</code> |
| <code>esp_event_handler_unregister_with()</code> | <code>esp_event_handler_unregister()</code> |
| <code>esp_event_post_to()</code> | <code>esp_event_post()</code> |

If you compare the signatures for both, they are mostly similar except the for the lack of loop handle specification for the default event loop APIs.

Other than the API difference and the special designation to which system events are posted to, there is no difference to how default event loops and user event loops behave. It is even possible for users to post their own events to the default event loop, should the user opt to not create their own loops to save memory.

Notes on Handler Registration

It is possible to register a single handler to multiple events individually, i.e. using multiple calls to [`esp_event_handler_register_with\(\)`](#). For those multiple calls, the specific event base and event ID can be specified with which the handler should execute.

However, in some cases it is desirable for a handler to execute on (1) all events that get posted to a loop or (2) all events of a particular base identifier. This is possible using the special event base identifier `ESP_EVENT_ANY_BASE` and special event ID `ESP_EVENT_ANY_ID`. These special identifiers may be passed as the event base and event ID arguments for [`esp_event_handler_register_with\(\)`](#).

Therefore, the valid arguments to [`esp_event_handler_register_with\(\)`](#) are:

1. <event base>, <event ID> - handler executes when the event with base <event base> and event ID <event ID> gets posted to the loop
2. <event base>, `ESP_EVENT_ANY_ID` - handler executes when any event with base <event base> gets posted to the loop
3. `ESP_EVENT_ANY_BASE`, `ESP_EVENT_ANY_ID` - handler executes when any event gets posted to the loop

As an example, suppose the following handler registrations were performed:

```

esp_event_handler_register_with(loop_handle, MY_EVENT_BASE, MY_EVENT_ID, run_on_
↳event_1, ...);
esp_event_handler_register_with(loop_handle, MY_EVENT_BASE, ESP_EVENT_ANY_ID, run_
↳on_event_2, ...);
esp_event_handler_register_with(loop_handle, ESP_EVENT_ANY_BASE, ESP_EVENT_ANY_ID,
↳run_on_event_3, ...);

```

If the hypothetical event `MY_EVENT_BASE, MY_EVENT_ID` is posted, all three handlers `run_on_event_1`, `run_on_event_2`, and `run_on_event_3` would execute.

If the hypothetical event `MY_EVENT_BASE, MY_OTHER_EVENT_ID` is posted, only `run_on_event_2` and `run_on_event_3` would execute.

If the hypothetical event `MY_OTHER_EVENT_BASE, MY_OTHER_EVENT_ID` is posted, only `run_on_event_3` would execute.

Handler Registration and Handler Dispatch Order The general rule is that for handlers that match a certain posted event during dispatch, those which are registered first also gets executed first. The user can then control which handlers get executed first by registering them before other handlers, provided that all registrations are performed using a single task. If the user plans to take advantage of this behavior, caution must be exercised if there are multiple tasks registering handlers. While the ‘first registered, first executed’ behavior still holds true, the task which gets executed first will also get their handlers registered first. Handlers registered one after the other by a single task will still be dispatched in the order relative to each other, but if that task gets pre-empted in between registration by another task which also registers handlers; then during dispatch those handlers will also get executed in between.

Event loop profiling

A configuration option `CONFIG_ESP_EVENT_LOOP_PROFILING` can be enabled in order to activate statistics collection for all event loops created. The function `esp_event_dump()` can be used to output the collected statistics to a file stream. More details on the information included in the dump can be found in the `esp_event_dump()` API Reference.

Application Example

Examples on using the `esp_event` library can be found in `system/esp_event`. The examples cover event declaration, loop creation, handler registration and unregistration and event posting.

Other examples which also adopt `esp_event` library:

- [NMEA Parser](#), which will decode the statements received from GPS.

API Reference

Header File

- `esp_event/include/esp_event.h`

Functions

```

esp_err_t esp_event_loop_create(const esp_event_loop_args_t *event_loop_args,
                               esp_event_loop_handle_t *event_loop)

```

Create a new event loop.

Return

- `ESP_OK`: Success
- `ESP_ERR_INVALID_ARG`: `event_loop_args` or `event_loop` was `NULL`
- `ESP_ERR_NO_MEM`: Cannot allocate memory for event loops list
- `ESP_FAIL`: Failed to create task loop

- Others: Fail

Parameters

- [in] `event_loop_args`: configuration structure for the event loop to create
- [out] `event_loop`: handle to the created event loop

esp_err_t **esp_event_loop_delete** (*esp_event_loop_handle_t* *event_loop*)

Delete an existing event loop.

Return

- ESP_OK: Success
- Others: Fail

Parameters

- [in] `event_loop`: event loop to delete, must not be NULL

esp_err_t **esp_event_loop_create_default** (void)

Create default event loop.

Return

- ESP_OK: Success
- ESP_ERR_NO_MEM: Cannot allocate memory for event loops list
- ESP_FAIL: Failed to create task loop
- Others: Fail

esp_err_t **esp_event_loop_delete_default** (void)

Delete the default event loop.

Return

- ESP_OK: Success
- Others: Fail

esp_err_t **esp_event_loop_run** (*esp_event_loop_handle_t* *event_loop*, TickType_t *ticks_to_run*)

Dispatch events posted to an event loop.

This function is used to dispatch events posted to a loop with no dedicated task, i.e task name was set to NULL in `event_loop_args` argument during loop creation. This function includes an argument to limit the amount of time it runs, returning control to the caller when that time expires (or some time afterwards). There is no guarantee that a call to this function will exit at exactly the time of expiry. There is also no guarantee that events have been dispatched during the call, as the function might have spent all of the allotted time waiting on the event queue. Once an event has been unqueued, however, it is guaranteed to be dispatched. This guarantee contributes to not being able to exit exactly at time of expiry as (1) blocking on internal mutexes is necessary for dispatching the unqueued event, and (2) during dispatch of the unqueued event there is no way to control the time occupied by handler code execution. The guaranteed time of exit is therefore the allotted time + amount of time required to dispatch the last unqueued event.

In cases where waiting on the queue times out, ESP_OK is returned and not ESP_ERR_TIMEOUT, since it is normal behavior.

Note encountering an unknown event that has been posted to the loop will only generate a warning, not an error.

Return

- ESP_OK: Success
- Others: Fail

Parameters

- [in] `event_loop`: event loop to dispatch posted events from, must not be NULL
- [in] `ticks_to_run`: number of ticks to run the loop

esp_err_t **esp_event_handler_register** (*esp_event_base_t* *event_base*, int32_t *event_id*,
esp_event_handler_t *event_handler*, void
**event_handler_arg*)

Register an event handler to the system event loop.

This function can be used to register a handler for either: (1) specific events, (2) all events of a certain event base, or (3) all events known by the system event loop.

- specific events: specify exact `event_base` and `event_id`

- all events of a certain base: specify exact `event_base` and use `ESP_EVENT_ANY_ID` as the `event_id`
- all events known by the loop: use `ESP_EVENT_ANY_BASE` for `event_base` and `ESP_EVENT_ANY_ID` as the `event_id`

Registering multiple handlers to events is possible. Registering a single handler to multiple events is also possible. However, registering the same handler to the same event multiple times would cause the previous registrations to be overwritten.

Note the event loop library does not maintain a copy of `event_handler_arg`, therefore the user should ensure that `event_handler_arg` still points to a valid location by the time the handler gets called

Return

- `ESP_OK`: Success
- `ESP_ERR_NO_MEM`: Cannot allocate memory for the handler
- `ESP_ERR_INVALID_ARG`: Invalid combination of event base and event id
- Others: Fail

Parameters

- `[in] event_base`: the base id of the event to register the handler for
- `[in] event_id`: the id of the event to register the handler for
- `[in] event_handler`: the handler function which gets called when the event is dispatched
- `[in] event_handler_arg`: data, aside from event data, that is passed to the handler when it is called

```
esp_err_t esp_event_handler_register_with(esp_event_loop_handle_t event_loop,  
                                         esp_event_base_t event_base, int32_t event_id,  
                                         esp_event_handler_t event_handler, void  
                                         *event_handler_arg)
```

Register an event handler to a specific loop.

This function behaves in the same manner as `esp_event_handler_register`, except the additional specification of the event loop to register the handler to.

Note the event loop library does not maintain a copy of `event_handler_arg`, therefore the user should ensure that `event_handler_arg` still points to a valid location by the time the handler gets called

Return

- `ESP_OK`: Success
- `ESP_ERR_NO_MEM`: Cannot allocate memory for the handler
- `ESP_ERR_INVALID_ARG`: Invalid combination of event base and event id
- Others: Fail

Parameters

- `[in] event_loop`: the event loop to register this handler function to, must not be `NULL`
- `[in] event_base`: the base id of the event to register the handler for
- `[in] event_id`: the id of the event to register the handler for
- `[in] event_handler`: the handler function which gets called when the event is dispatched
- `[in] event_handler_arg`: data, aside from event data, that is passed to the handler when it is called

```
esp_err_t esp_event_handler_unregister(esp_event_base_t event_base, int32_t event_id,  
                                       esp_event_handler_t event_handler)
```

Unregister a handler with the system event loop.

Unregisters a handler so it will no longer be called during dispatch. Handlers can be unregistered for any combination of `event_base` and `event_id` which were previously registered. To unregister a handler, the `event_base` and `event_id` arguments must match exactly the arguments passed to `esp_event_handler_register()` when that handler was registered. Passing `ESP_EVENT_ANY_BASE` and/or `ESP_EVENT_ANY_ID` will only unregister handlers that were registered with the same wildcard arguments.

Note When using `ESP_EVENT_ANY_ID`, handlers registered to specific event IDs using the same base will not be unregistered. When using `ESP_EVENT_ANY_BASE`, events registered to specific bases will also not be unregistered. This avoids accidental unregistration of handlers registered by other users or components.

Return `ESP_OK` success

Return `ESP_ERR_INVALID_ARG` invalid combination of event base and event id

Return others fail

Parameters

- [in] `event_base`: the base of the event with which to unregister the handler
- [in] `event_id`: the id of the event with which to unregister the handler
- [in] `event_handler`: the handler to unregister

esp_err_t **esp_event_handler_unregister_with** (*esp_event_loop_handle_t* `event_loop`,
esp_event_base_t `event_base`, *int32_t* `event_id`,
esp_event_handler_t `event_handler`)

Unregister a handler with the system event loop.

This function behaves in the same manner as `esp_event_handler_unregister`, except the additional specification of the event loop to unregister the handler with.

Return

- `ESP_OK`: Success
- `ESP_ERR_INVALID_ARG`: Invalid combination of event base and event id
- Others: Fail

Parameters

- [in] `event_loop`: the event loop with which to unregister this handler function, must not be NULL
- [in] `event_base`: the base of the event with which to unregister the handler
- [in] `event_id`: the id of the event with which to unregister the handler
- [in] `event_handler`: the handler to unregister

esp_err_t **esp_event_post** (*esp_event_base_t* `event_base`, *int32_t* `event_id`, *void *event_data*, *size_t*
`event_data_size`, *TickType_t* `ticks_to_wait`)

Posts an event to the system default event loop. The event loop library keeps a copy of `event_data` and manages the copy's lifetime automatically (allocation + deletion); this ensures that the data the handler receives is always valid.

Return

- `ESP_OK`: Success
- `ESP_ERR_TIMEOUT`: Time to wait for event queue to unblock expired, queue full when posting from ISR
- `ESP_ERR_INVALID_ARG`: Invalid combination of event base and event id
- Others: Fail

Parameters

- [in] `event_base`: the event base that identifies the event
- [in] `event_id`: the event id that identifies the event
- [in] `event_data`: the data, specific to the event occurrence, that gets passed to the handler
- [in] `event_data_size`: the size of the event data
- [in] `ticks_to_wait`: number of ticks to block on a full event queue

esp_err_t **esp_event_post_to** (*esp_event_loop_handle_t* `event_loop`, *esp_event_base_t* `event_base`,
int32_t `event_id`, *void *event_data*, *size_t* `event_data_size`, *TickType_t*
`ticks_to_wait`)

Posts an event to the specified event loop. The event loop library keeps a copy of `event_data` and manages the copy's lifetime automatically (allocation + deletion); this ensures that the data the handler receives is always valid.

This function behaves in the same manner as `esp_event_post_to`, except the additional specification of the event loop to post the event to.

Return

- `ESP_OK`: Success
- `ESP_ERR_TIMEOUT`: Time to wait for event queue to unblock expired, queue full when posting from ISR
- `ESP_ERR_INVALID_ARG`: Invalid combination of event base and event id
- Others: Fail

Parameters

- [in] `event_loop`: the event loop to post to, must not be NULL
- [in] `event_base`: the event base that identifies the event

- [in] `event_id`: the event id that identifies the event
- [in] `event_data`: the data, specific to the event occurrence, that gets passed to the handler
- [in] `event_data_size`: the size of the event data
- [in] `ticks_to_wait`: number of ticks to block on a full event queue

`esp_err_t esp_event_dump` (FILE **file*)

Dumps statistics of all event loops.

Dumps event loop info in the format:

```

event loop
    handler
    handler
    ...
event loop
    handler
    handler
    ...

where:

event loop
    format: address,name rx:total_recieved dr:total_dropped
    where:
        address - memory address of the event loop
        name - name of the event loop, 'none' if no dedicated task
        total_recieved - number of successfully posted events
        total_dropped - number of events unsuccessfully posted due to queue
↳being full

handler
    format: address ev:base,id inv:total_invoked run:total_runtime
    where:
        address - address of the handler function
        base,id - the event specified by event base and id this handler
↳executes
        total_invoked - number of times this handler has been invoked
        total_runtime - total amount of time used for invoking this handler

```

Note this function is a noop when `CONFIG_ESP_EVENT_LOOP_PROFILING` is disabled

Return

- `ESP_OK`: Success
- `ESP_ERR_NO_MEM`: Cannot allocate memory for event loops list
- Others: Fail

Parameters

- [in] `file`: the file stream to output to

Structures

struct esp_event_loop_args_t

Configuration for creating event loops.

Public Members

`int32_t queue_size`

size of the event loop queue

`const char *task_name`

name of the event loop task; if NULL, a dedicated task is not created for event loop

`UBaseType_t task_priority`

priority of the event loop task, ignored if task name is NULL

uint32_t **task_stack_size**

stack size of the event loop task, ignored if task name is NULL

BaseType_t **task_core_id**

core to which the event loop task is pinned to, ignored if task name is NULL

Header File

- [esp_event/include/esp_event_base.h](#)

Macros

ESP_EVENT_DECLARE_BASE (id)

ESP_EVENT_DEFINE_BASE (id)

ESP_EVENT_ANY_BASE

register handler for any event base

ESP_EVENT_ANY_ID

register handler for any event id

Type Definitions

typedef const char ***esp_event_base_t**

unique pointer to a subsystem that exposes events

typedef void ***esp_event_loop_handle_t**

a number that identifies an event with respect to a base

typedef void (***esp_event_handler_t**) (void *event_handler_arg, *esp_event_base_t* event_base, int32_t event_id, void *event_data)

function called when an event is posted to the queue

Related Documents

Legacy event loop

API Reference

Header File

- [esp_event/include/esp_event_legacy.h](#)

Functions

esp_err_t **esp_event_send** (*system_event_t* *event)

Send a event to event task.

Other task/modules, such as the tcpip_adapter, can call this API to send an event to event task

Note This API is part of the legacy event system. New code should use event library API in esp_event.h

Return ESP_OK : succeed

Return others : fail

Parameters

- event: Event to send

esp_err_t **esp_event_send_internal** (*esp_event_base_t* event_base, int32_t event_id, void *event_data, size_t event_data_size, TickType_t ticks_to_wait)

Send a event to event task.

Other task/modules, such as the tcpip_adapter, can call this API to send an event to event task

Note This API is used by WiFi Driver only.

Return ESP_OK : succeed

Return others : fail

Parameters

- [in] event_base: the event base that identifies the event
- [in] event_id: the event id that identifies the event
- [in] event_data: the data, specific to the event occurrence, that gets passed to the handler
- [in] event_data_size: the size of the event data
- [in] ticks_to_wait: number of ticks to block on a full event queue

esp_err_t **esp_event_process_default** (*system_event_t* *event)

Default event handler for system events.

This function performs default handling of system events. When using esp_event_loop APIs, it is called automatically before invoking the user-provided callback function.

Note This API is part of the legacy event system. New code should use event library API in esp_event.h

Applications which implement a custom event loop must call this function as part of event processing.

Return ESP_OK if an event was handled successfully

Parameters

- event: pointer to event to be handled

void **esp_event_set_default_eth_handlers** (void)

Install default event handlers for Ethernet interface.

Note This API is part of the legacy event system. New code should use event library API in esp_event.h

void **esp_event_set_default_wifi_handlers** (void)

Install default event handlers for Wi-Fi interfaces (station and AP)

Note This API is part of the legacy event system. New code should use event library API in esp_event.h

esp_err_t **esp_event_loop_init** (*system_event_cb_t* cb, void *ctx)

Initialize event loop.

Create the event handler and task

Note This API is part of the legacy event system. New code should use event library API in esp_event.h

Return

- ESP_OK: succeed
- others: fail

Parameters

- cb: application specified event callback, it can be modified by call esp_event_set_cb
- ctx: reserved for user

system_event_cb_t **esp_event_loop_set_cb** (*system_event_cb_t* cb, void *ctx)

Set application specified event callback function.

Note This API is part of the legacy event system. New code should use event library API in esp_event.h

Attention 1. If cb is NULL, means application don't need to handle If cb is not NULL, it will be call when an event is received, after the default event callback is completed

Return old callback

Parameters

- cb: application callback function
- ctx: argument to be passed to callback

Unions

union **system_event_info_t**

#include <esp_event_legacy.h> Union of all possible system_event argument structures

Public Members

- system_event_sta_connected_t* **connected**
ESP32 station connected to AP
- system_event_sta_disconnected_t* **disconnected**
ESP32 station disconnected to AP
- system_event_sta_scan_done_t* **scan_done**
ESP32 station scan (APs) done
- system_event_sta_authmode_change_t* **auth_change**
the auth mode of AP ESP32 station connected to changed
- system_event_sta_got_ip_t* **got_ip**
ESP32 station got IP, first time got IP or when IP is changed
- system_event_sta_wps_er_pin_t* **sta_er_pin**
ESP32 station WPS enrollee mode PIN code received
- system_event_sta_wps_fail_reason_t* **sta_er_fail_reason**
ESP32 station WPS enrollee mode failed reason code received
- system_event_sta_wps_er_success_t* **sta_er_success**
ESP32 station WPS enrollee success
- system_event_ap_staconnected_t* **sta_connected**
a station connected to ESP32 soft-AP
- system_event_ap_stadisconnected_t* **sta_disconnected**
a station disconnected to ESP32 soft-AP
- system_event_ap_probe_req_rx_t* **ap_probereqrecved**
ESP32 soft-AP receive probe request packet
- system_event_ap_staipassigned_t* **ap_staipassigned**
ESP32 soft-AP assign an IP to the station
- system_event_got_ip6_t* **got_ip6**
ESP32 station or ap or ethernet ipv6 addr state change to preferred

Structures

struct system_event_t
Event, as a tagged enum

Public Members

- system_event_id_t* **event_id**
event ID
- system_event_info_t* **event_info**
event information

Macros

SYSTEM_EVENT_AP_STA_GOT_IP6

Type Definitions

typedef *wifi_event_sta_wps_fail_reason_t* **system_event_sta_wps_fail_reason_t**
Argument structure of SYSTEM_EVENT_STA_WPS_ER_FAILED event

typedef *wifi_event_sta_scan_done_t* **system_event_sta_scan_done_t**
Argument structure of SYSTEM_EVENT_SCAN_DONE event

```

typedef wifi_event_sta_connected_t system_event_sta_connected_t
    Argument structure of SYSTEM_EVENT_STA_CONNECTED event

typedef wifi_event_sta_disconnected_t system_event_sta_disconnected_t
    Argument structure of SYSTEM_EVENT_STA_DISCONNECTED event

typedef wifi_event_sta_authmode_change_t system_event_sta_authmode_change_t
    Argument structure of SYSTEM_EVENT_STA_AUTHMODE_CHANGE event

typedef wifi_event_sta_wps_er_pin_t system_event_sta_wps_er_pin_t
    Argument structure of SYSTEM_EVENT_STA_WPS_ER_PIN event

typedef wifi_event_sta_wps_er_success_t system_event_sta_wps_er_success_t
    Argument structure of SYSTEM_EVENT_STA_WPS_ER_PIN event

typedef wifi_event_ap_staconnected_t system_event_ap_staconnected_t
    Argument structure of event

typedef wifi_event_ap_stadisconnected_t system_event_ap_stadisconnected_t
    Argument structure of event

typedef wifi_event_ap_probe_req_rx_t system_event_ap_probe_req_rx_t
    Argument structure of event

typedef ip_event_ap_staassigned_t system_event_ap_staassigned_t
    Argument structure of event

typedef ip_event_got_ip_t system_event_sta_got_ip_t
    Argument structure of event

typedef ip_event_got_ip6_t system_event_got_ip6_t
    Argument structure of event

typedef esp_err_t (*system_event_handler_t) (esp_event_base_t event_base, int32_t event_id,
    void *event_data, size_t event_data_size, Tick-
    Type_t ticks_to_wait)

    Event handler function type

typedef esp_err_t (*system_event_cb_t) (void *ctx, system_event_t *event)
    Application specified event callback function.

```

Note This API is part of the legacy event system. New code should use event library API in esp_event.h

Return

- ESP_OK: succeed
- others: fail

Parameters

- ctx: reserved for user
- event: event type defined in this file

Enumerations

```

enum system_event_id_t
    System event types enumeration

```

Values:

```

SYSTEM_EVENT_WIFI_READY = 0
    ESP32 WiFi ready

```

```

SYSTEM_EVENT_SCAN_DONE
    ESP32 finish scanning AP

```

```

SYSTEM_EVENT_STA_START
    ESP32 station start

```

```

SYSTEM_EVENT_STA_STOP
    ESP32 station stop

```


SYSTEM_EVENT_STA_CONNECTED
ESP32 station connected to AP

SYSTEM_EVENT_STA_DISCONNECTED
ESP32 station disconnected from AP

SYSTEM_EVENT_STA_AUTHMODE_CHANGE
the auth mode of AP connected by ESP32 station changed

SYSTEM_EVENT_STA_GOT_IP
ESP32 station got IP from connected AP

SYSTEM_EVENT_STA_LOST_IP
ESP32 station lost IP and the IP is reset to 0

SYSTEM_EVENT_STA_BEACON_TIMEOUT
ESP32 station beacon timeout

SYSTEM_EVENT_STA_WPS_ER_SUCCESS
ESP32 station wps succeeds in enrollee mode

SYSTEM_EVENT_STA_WPS_ER_FAILED
ESP32 station wps fails in enrollee mode

SYSTEM_EVENT_STA_WPS_ER_TIMEOUT
ESP32 station wps timeout in enrollee mode

SYSTEM_EVENT_STA_WPS_ER_PIN
ESP32 station wps pin code in enrollee mode

SYSTEM_EVENT_STA_WPS_ER_PBC_OVERLAP
ESP32 station wps overlap in enrollee mode

SYSTEM_EVENT_AP_START
ESP32 soft-AP start

SYSTEM_EVENT_AP_STOP
ESP32 soft-AP stop

SYSTEM_EVENT_AP_STA_CONNECTED
a station connected to ESP32 soft-AP

SYSTEM_EVENT_AP_STA_DISCONNECTED
a station disconnected from ESP32 soft-AP

SYSTEM_EVENT_AP_STA_IP_ASSIGNED
ESP32 soft-AP assign an IP to a connected station

SYSTEM_EVENT_AP_PROBREQ_RECV
Receive probe request packet in soft-AP interface

SYSTEM_EVENT_GOT_IP6
ESP32 station or ap or ethernet interface v6IP addr is preferred

SYSTEM_EVENT_ETH_START
ESP32 ethernet start

SYSTEM_EVENT_ETH_STOP
ESP32 ethernet stop

SYSTEM_EVENT_ETH_CONNECTED
ESP32 ethernet phy link up

SYSTEM_EVENT_ETH_DISCONNECTED
ESP32 ethernet phy link down

SYSTEM_EVENT_ETH_GOT_IP
ESP32 ethernet got IP from connected AP

SYSTEM_EVENT_MAX

Number of members in this enum

2.7.8 FreeRTOS

Overview

This section contains documentation of FreeRTOS types, functions, and macros. It is automatically generated from FreeRTOS header files.

Note: ESP-IDF FreeRTOS is based on the Xtensa port of FreeRTOS v8.2.0, however some functions of FreeRTOS v9.0.0 have been backported. See the [Backported Features](#) for more information.

For more information about FreeRTOS features specific to ESP-IDF, see [ESP-IDF FreeRTOS SMP Changes](#) and [ESP-IDF FreeRTOS Additions](#).

Task API

Header File

- [freertos/include/freertos/task.h](#)

Functions

`BaseType_t xTaskCreatePinnedToCore (TaskFunction_t pvTaskCode, const char *const pcName, const uint32_t usStackDepth, void *const pvParameters, UBaseType_t uxPriority, TaskHandle_t *const pvCreatedTask, const BaseType_t xCoreID)`

Create a new task with a specified affinity.

This function is similar to `xTaskCreate`, but allows setting task affinity in SMP system.

Return `pdPASS` if the task was successfully created and added to a ready list, otherwise an error code defined in the file `projdefs.h`

Parameters

- `pvTaskCode`: Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop).
- `pcName`: A descriptive name for the task. This is mainly used to facilitate debugging. Max length defined by `configMAX_TASK_NAME_LEN` - default is 16.
- `usStackDepth`: The size of the task stack specified as the number of bytes. Note that this differs from vanilla FreeRTOS.
- `pvParameters`: Pointer that will be used as the parameter for the task being created.
- `uxPriority`: The priority at which the task should run. Systems that include MPU support can optionally create tasks in a privileged (system) mode by setting bit `portPRIVILEGE_BIT` of the priority parameter. For example, to create a privileged task at priority 2 the `uxPriority` parameter should be set to `(2 | portPRIVILEGE_BIT)`.
- `pvCreatedTask`: Used to pass back a handle by which the created task can be referenced.
- `xCoreID`: If the value is `tskNO_AFFINITY`, the created task is not pinned to any CPU, and the scheduler can run it on any core available. Values 0 or 1 indicate the index number of the CPU which the task should be pinned to. Specifying values larger than `(portNUM_PROCESSORS - 1)` will cause the function to fail.

`static BaseType_t xTaskCreate (TaskFunction_t pvTaskCode, const char *const pcName, const uint32_t usStackDepth, void *const pvParameters, UBaseType_t uxPriority, TaskHandle_t *const pvCreatedTask)`

Create a new task and add it to the list of tasks that are ready to run.

Internally, within the FreeRTOS implementation, tasks use two blocks of memory. The first block is used to hold the task's data structures. The second block is used by the task as its stack. If a task is created using

`xTaskCreate()` then both blocks of memory are automatically dynamically allocated inside the `xTaskCreate()` function. (see <http://www.freertos.org/a00111.html>). If a task is created using `xTaskCreateStatic()` then the application writer must provide the required memory. `xTaskCreateStatic()` therefore allows a task to be created without using any dynamic memory allocation.

See `xTaskCreateStatic()` for a version that does not use any dynamic memory allocation.

`xTaskCreate()` can only be used to create a task that has unrestricted access to the entire microcontroller memory map. Systems that include MPU support can alternatively create an MPU constrained task using `xTaskCreateRestricted()`.

Example usage:

```
// Task to be created.
void vTaskCode( void * pvParameters )
{
    for( ;; )
    {
        // Task code goes here.
    }
}

// Function that creates a task.
void vOtherFunction( void )
{
    static uint8_t ucParameterToPass;
    TaskHandle_t xHandle = NULL;

    // Create the task, storing the handle. Note that the passed parameter
    ↪ucParameterToPass
    // must exist for the lifetime of the task, so in this case is declared
    ↪static. If it was just an
    // an automatic stack variable it might no longer exist, or at least have
    ↪been corrupted, by the time
    // the new task attempts to access it.
    xTaskCreate( vTaskCode, "NAME", STACK_SIZE, &ucParameterToPass, tskIDLE_
    ↪PRIORITY, &xHandle );
    configASSERT( xHandle );

    // Use the handle to delete the task.
    if( xHandle != NULL )
    {
        vTaskDelete( xHandle );
    }
}
```

Return `pdPASS` if the task was successfully created and added to a ready list, otherwise an error code defined in the file `projdefs.h`

Note If program uses thread local variables (ones specified with “`__thread`” keyword) then storage for them will be allocated on the task’s stack.

Parameters

- `pvTaskCode`: Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop).
- `pcName`: A descriptive name for the task. This is mainly used to facilitate debugging. Max length defined by `configMAX_TASK_NAME_LEN` - default is 16.
- `usStackDepth`: The size of the task stack specified as the number of bytes. Note that this differs from vanilla FreeRTOS.
- `pvParameters`: Pointer that will be used as the parameter for the task being created.
- `uxPriority`: The priority at which the task should run. Systems that include MPU support can optionally create tasks in a privileged (system) mode by setting bit `portPRIVILEGE_BIT` of the priority parameter. For example, to create a privileged task at priority 2 the `uxPriority` parameter should be set to `(2 | portPRIVILEGE_BIT)`.

- `pvCreatedTask`: Used to pass back a handle by which the created task can be referenced.

TaskHandle_t `xTaskCreateStaticPinnedToCore` (TaskFunction_t *pvTaskCode*, **const** char ***const** *pcName*, **const** uint32_t *ulStackDepth*, void ***const** *pvParameters*, UBaseType_t *uxPriority*, StackType_t ***const** *pxStackBuffer*, StaticTask_t ***const** *pxTaskBuffer*, **const** BaseType_t *xCoreID*)

Create a new task with a specified affinity.

This function is similar to `xTaskCreateStatic`, but allows specifying task affinity in an SMP system.

Return If neither `pxStackBuffer` or `pxTaskBuffer` are NULL, then the task will be created and a task handle will be returned by which the created task can be referenced. If either `pxStackBuffer` or `pxTaskBuffer` are NULL then the task will not be created and NULL is returned.

Parameters

- `pvTaskCode`: Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop).
- `pcName`: A descriptive name for the task. This is mainly used to facilitate debugging. The maximum length of the string is defined by `configMAX_TASK_NAME_LEN` in `FreeRTOSConfig.h`.
- `ulStackDepth`: The size of the task stack specified as the number of bytes. Note that this differs from vanilla FreeRTOS.
- `pvParameters`: Pointer that will be used as the parameter for the task being created.
- `uxPriority`: The priority at which the task will run.
- `pxStackBuffer`: Must point to a `StackType_t` array that has at least `ulStackDepth` indexes - the array will then be used as the task's stack, removing the need for the stack to be allocated dynamically.
- `pxTaskBuffer`: Must point to a variable of type `StaticTask_t`, which will then be used to hold the task's data structures, removing the need for the memory to be allocated dynamically.
- `xCoreID`: If the value is `tskNO_AFFINITY`, the created task is not pinned to any CPU, and the scheduler can run it on any core available. Values 0 or 1 indicate the index number of the CPU which the task should be pinned to. Specifying values larger than `(portNUM_PROCESSORS - 1)` will cause the function to fail.

static TaskHandle_t `xTaskCreateStatic` (TaskFunction_t *pvTaskCode*, **const** char ***const** *pcName*, **const** uint32_t *ulStackDepth*, void ***const** *pvParameters*, UBaseType_t *uxPriority*, StackType_t ***const** *pxStackBuffer*, StaticTask_t ***const** *pxTaskBuffer*)

Create a new task and add it to the list of tasks that are ready to run.

Internally, within the FreeRTOS implementation, tasks use two blocks of memory. The first block is used to hold the task's data structures. The second block is used by the task as its stack. If a task is created using `xTaskCreate()` then both blocks of memory are automatically dynamically allocated inside the `xTaskCreate()` function. (see <http://www.freertos.org/a00111.html>). If a task is created using `xTaskCreateStatic()` then the application writer must provide the required memory. `xTaskCreateStatic()` therefore allows a task to be created without using any dynamic memory allocation.

Example usage:

```
// Dimensions the buffer that the task being created will use as its stack.
// NOTE: This is the number of bytes the stack will hold, not the number of
// words as found in vanilla FreeRTOS.
#define STACK_SIZE 200

// Structure that will hold the TCB of the task being created.
StaticTask_t xTaskBuffer;

// Buffer that the task being created will use as its stack. Note this is
// an array of StackType_t variables. The size of StackType_t is dependent on
// the RTOS port.
StackType_t xStack[ STACK_SIZE ];
```

(continues on next page)

```

// Function that implements the task being created.
void vTaskCode( void * pvParameters )
{
    // The parameter value is expected to be 1 as 1 is passed in the
    // pvParameters value in the call to xTaskCreateStatic().
    configASSERT( ( uint32_t ) pvParameters == 1UL );

    for( ;; )
    {
        // Task code goes here.
    }
}

// Function that creates a task.
void vOtherFunction( void )
{
    TaskHandle_t xHandle = NULL;

    // Create the task without using any dynamic memory allocation.
    xHandle = xTaskCreateStatic(
        vTaskCode,           // Function that implements the task.
        "NAME",             // Text name for the task.
        STACK_SIZE,        // Stack size in bytes, not words.
        ( void * ) 1,      // Parameter passed into the task.
        tskIDLE_PRIORITY, // Priority at which the task is created.
        xStack,            // Array to use as the task's stack.
        &xTaskBuffer );    // Variable to hold the task's data_
↪structure.

    // puxStackBuffer and pxTaskBuffer were not NULL, so the task will have
    // been created, and xHandle will be the task's handle. Use the handle
    // to suspend the task.
    vTaskSuspend( xHandle );
}

```

Return If neither pxStackBuffer or pxTaskBuffer are NULL, then the task will be created and a task handle will be returned by which the created task can be referenced. If either pxStackBuffer or pxTaskBuffer are NULL then the task will not be created and NULL is returned.

Note If program uses thread local variables (ones specified with “__thread” keyword) then storage for them will be allocated on the task’s stack.

Parameters

- pvTaskCode: Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop).
- pcName: A descriptive name for the task. This is mainly used to facilitate debugging. The maximum length of the string is defined by configMAX_TASK_NAME_LEN in FreeRTOSConfig.h.
- ulStackDepth: The size of the task stack specified as the number of bytes. Note that this differs from vanilla FreeRTOS.
- pvParameters: Pointer that will be used as the parameter for the task being created.
- uxPriority: The priority at which the task will run.
- pxStackBuffer: Must point to a StackType_t array that has at least ulStackDepth indexes - the array will then be used as the task’s stack, removing the need for the stack to be allocated dynamically.
- pxTaskBuffer: Must point to a variable of type StaticTask_t, which will then be used to hold the task’s data structures, removing the need for the memory to be allocated dynamically.

void **vTaskDelete** (*TaskHandle_t xTaskToDelete*)

Remove a task from the RTOS real time kernel’s management.

The task being deleted will be removed from all ready, blocked, suspended and event lists.

INCLUDE_vTaskDelete must be defined as 1 for this function to be available. See the configuration section for more information.

See the demo application file death.c for sample code that utilises vTaskDelete ().

Note The idle task is responsible for freeing the kernel allocated memory from tasks that have been deleted. It is therefore important that the idle task is not starved of microcontroller processing time if your application makes any calls to vTaskDelete (). Memory allocated by the task code is not automatically freed, and should be freed before the task is deleted.

Example usage:

```
void vOtherFunction( void )
{
    TaskHandle_t xHandle;

    // Create the task, storing the handle.
    xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &
    ↪xHandle );

    // Use the handle to delete the task.
    vTaskDelete( xHandle );
}
```

Parameters

- `xTaskToDelete`: The handle of the task to be deleted. Passing NULL will cause the calling task to be deleted.

void **vTaskDelay** (const TickType_t *xTicksToDelay*)

Delay a task for a given number of ticks.

The actual time that the task remains blocked depends on the tick rate. The constant portTICK_PERIOD_MS can be used to calculate real time from the tick rate - with the resolution of one tick period.

INCLUDE_vTaskDelay must be defined as 1 for this function to be available. See the configuration section for more information.

vTaskDelay() specifies a time at which the task wishes to unblock relative to the time at which vTaskDelay() is called. For example, specifying a block period of 100 ticks will cause the task to unblock 100 ticks after vTaskDelay() is called. vTaskDelay() does not therefore provide a good method of controlling the frequency of a periodic task as the path taken through the code, as well as other task and interrupt activity, will effect the frequency at which vTaskDelay() gets called and therefore the time at which the task next executes. See vTaskDelayUntil() for an alternative API function designed to facilitate fixed frequency execution. It does this by specifying an absolute time (rather than a relative time) at which the calling task should unblock.

Example usage:

```
void vTaskFunction( void * pvParameters )
{
    // Block for 500ms.
    const TickType_t xDelay = 500 / portTICK_PERIOD_MS;

    for ( ;; )
    {
        // Simply toggle the LED every 500ms, blocking between each toggle.
        vToggleLED();
        vTaskDelay( xDelay );
    }
}
```

Parameters

- `xTicksToDelay`: The amount of time, in tick periods, that the calling task should block.

void **vTaskDelayUntil** (TickType_t *const *pxPreviousWakeTime*, const TickType_t *xTimeIncrement*)

Delay a task until a specified time.

INCLUDE_vTaskDelayUntil must be defined as 1 for this function to be available. See the configuration section for more information.

This function can be used by periodic tasks to ensure a constant execution frequency.

This function differs from vTaskDelay () in one important aspect: vTaskDelay () will cause a task to block for the specified number of ticks from the time vTaskDelay () is called. It is therefore difficult to use vTaskDelay () by itself to generate a fixed execution frequency as the time between a task starting to execute and that task calling vTaskDelay () may not be fixed [the task may take a different path though the code between calls, or may get interrupted or preempted a different number of times each time it executes].

Whereas vTaskDelay () specifies a wake time relative to the time at which the function is called, vTaskDelayUntil () specifies the absolute (exact) time at which it wishes to unblock.

The constant portTICK_PERIOD_MS can be used to calculate real time from the tick rate - with the resolution of one tick period.

Example usage:

```
// Perform an action every 10 ticks.
void vTaskFunction( void * pvParameters )
{
    TickType_t xLastWakeTime;
    const TickType_t xFrequency = 10;

    // Initialise the xLastWakeTime variable with the current time.
    xLastWakeTime = xTaskGetTickCount ();
    for( ;; )
    {
        // Wait for the next cycle.
        vTaskDelayUntil( &xLastWakeTime, xFrequency );

        // Perform action here.
    }
}
```

Parameters

- *pxPreviousWakeTime*: Pointer to a variable that holds the time at which the task was last unblocked. The variable must be initialised with the current time prior to its first use (see the example below). Following this the variable is automatically updated within vTaskDelayUntil ().
- *xTimeIncrement*: The cycle time period. The task will be unblocked at time **pxPreviousWakeTime* + *xTimeIncrement*. Calling vTaskDelayUntil with the same *xTimeIncrement* parameter value will cause the task to execute with a fixed interface period.

UBaseType_t **uxTaskPriorityGet** (*TaskHandle_t xTask*)

Obtain the priority of any task.

INCLUDE_uxTaskPriorityGet must be defined as 1 for this function to be available. See the configuration section for more information.

Example usage:

```
void vAFunction( void )
{
    TaskHandle_t xHandle;

    // Create a task, storing the handle.
    xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle);
}
```

(continues on next page)

(continued from previous page)

```

// ...

// Use the handle to obtain the priority of the created task.
// It was created with tskIDLE_PRIORITY, but may have changed
// it itself.
if( uxTaskPriorityGet( xHandle ) != tskIDLE_PRIORITY )
{
    // The task has changed it's priority.
}

// ...

// Is our priority higher than the created task?
if( uxTaskPriorityGet( xHandle ) < uxTaskPriorityGet( NULL ) )
{
    // Our priority (obtained using NULL handle) is higher.
}
}

```

Return The priority of xTask.

Parameters

- xTask: Handle of the task to be queried. Passing a NULL handle results in the priority of the calling task being returned.

UBaseType_t **uxTaskPriorityGetFromISR** (*TaskHandle_t* xTask)

A version of uxTaskPriorityGet() that can be used from an ISR.

Return The priority of xTask.

Parameters

- xTask: Handle of the task to be queried. Passing a NULL handle results in the priority of the calling task being returned.

eTaskState **eTaskGetState** (*TaskHandle_t* xTask)

Obtain the state of any task.

States are encoded by the eTaskState enumerated type.

INCLUDE_eTaskGetState must be defined as 1 for this function to be available. See the configuration section for more information.

Return The state of xTask at the time the function was called. Note the state of the task might change between the function being called, and the functions return value being tested by the calling task.

Parameters

- xTask: Handle of the task to be queried.

void **vTaskPrioritySet** (*TaskHandle_t* xTask, UBaseType_t uxNewPriority)

Set the priority of any task.

INCLUDE_vTaskPrioritySet must be defined as 1 for this function to be available. See the configuration section for more information.

A context switch will occur before the function returns if the priority being set is higher than the currently executing task.

Example usage:

```

void vAFunction( void )
{
    TaskHandle_t xHandle;

    // Create a task, storing the handle.
    xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle
    ↪ );
}

```

(continues on next page)

(continued from previous page)

```

// ...

// Use the handle to raise the priority of the created task.
vTaskPrioritySet( xHandle, tskIDLE_PRIORITY + 1 );

// ...

// Use a NULL handle to raise our priority to the same value.
vTaskPrioritySet( NULL, tskIDLE_PRIORITY + 1 );
}

```

Parameters

- `xTask`: Handle to the task for which the priority is being set. Passing a NULL handle results in the priority of the calling task being set.
- `uxNewPriority`: The priority to which the task will be set.

void **vTaskSuspend** (*TaskHandle_t* *xTaskToSuspend*)

Suspend a task.

INCLUDE_vTaskSuspend must be defined as 1 for this function to be available. See the configuration section for more information.

When suspended, a task will never get any microcontroller processing time, no matter what its priority.

Calls to vTaskSuspend are not accumulative - i.e. calling vTaskSuspend () twice on the same task still only requires one call to vTaskResume () to ready the suspended task.

Example usage:

```

void vAFunction( void )
{
TaskHandle_t xHandle;

// Create a task, storing the handle.
xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle_
↪);

// ...

// Use the handle to suspend the created task.
vTaskSuspend( xHandle );

// ...

// The created task will not run during this period, unless
// another task calls vTaskResume( xHandle ).

//...

// Suspend ourselves.
vTaskSuspend( NULL );

// We cannot get here unless another task calls vTaskResume
// with our handle as the parameter.
}

```

Parameters

- `xTaskToSuspend`: Handle to the task being suspended. Passing a NULL handle will cause the calling task to be suspended.

void **vTaskResume** (*TaskHandle_t* *xTaskToResume*)

Resumes a suspended task.

INCLUDE_vTaskSuspend must be defined as 1 for this function to be available. See the configuration section for more information.

A task that has been suspended by one or more calls to vTaskSuspend () will be made available for running again by a single call to vTaskResume ().

Example usage:

```
void vAFunction( void )
{
    TaskHandle_t xHandle;

    // Create a task, storing the handle.
    xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle_
    → );

    // ...

    // Use the handle to suspend the created task.
    vTaskSuspend( xHandle );

    // ...

    // The created task will not run during this period, unless
    // another task calls vTaskResume( xHandle ).

    //...

    // Resume the suspended task ourselves.
    vTaskResume( xHandle );

    // The created task will once again get microcontroller processing
    // time in accordance with its priority within the system.
}
```

Parameters

- xTaskToResume: Handle to the task being readied.

BaseType_t **xTaskResumeFromISR** (*TaskHandle_t xTaskToResume*)

An implementation of vTaskResume() that can be called from within an ISR.

INCLUDE_xTaskResumeFromISR must be defined as 1 for this function to be available. See the configuration section for more information.

A task that has been suspended by one or more calls to vTaskSuspend () will be made available for running again by a single call to xTaskResumeFromISR ().

xTaskResumeFromISR() should not be used to synchronise a task with an interrupt if there is a chance that the interrupt could arrive prior to the task being suspended - as this can lead to interrupts being missed. Use of a semaphore as a synchronisation mechanism would avoid this eventuality.

Return pdTRUE if resuming the task should result in a context switch, otherwise pdFALSE. This is used by the ISR to determine if a context switch may be required following the ISR.

Parameters

- xTaskToResume: Handle to the task being readied.

void **vTaskSuspendAll** (void)

Suspends the scheduler without disabling interrupts.

Context switches will not occur while the scheduler is suspended.

After calling vTaskSuspendAll () the calling task will continue to execute without risk of being swapped out until a call to xTaskResumeAll () has been made.

API functions that have the potential to cause a context switch (for example, `vTaskDelayUntil()`, `xQueueSend()`, etc.) must not be called while the scheduler is suspended.

Example usage:

```
void vTask1( void * pvParameters )
{
    for( ;; )
    {
        // Task code goes here.

        // ...

        // At some point the task wants to perform a long operation during
        // which it does not want to get swapped out. It cannot use
        // taskENTER_CRITICAL ()/taskEXIT_CRITICAL () as the length of the
        // operation may cause interrupts to be missed - including the
        // ticks.

        // Prevent the real time kernel swapping out the task.
        vTaskSuspendAll ();

        // Perform the operation here. There is no need to use critical
        // sections as we have all the microcontroller processing time.
        // During this time interrupts will still operate and the kernel
        // tick count will be maintained.

        // ...

        // The operation is complete. Restart the kernel.
        xTaskResumeAll ();
    }
}
```

BaseType_t **xTaskResumeAll** (void)

Resumes scheduler activity after it was suspended by a call to `vTaskSuspendAll()`.

`xTaskResumeAll()` only resumes the scheduler. It does not unsuspend tasks that were previously suspended by a call to `vTaskSuspend()`.

Example usage:

```
void vTask1( void * pvParameters )
{
    for( ;; )
    {
        // Task code goes here.

        // ...

        // At some point the task wants to perform a long operation during
        // which it does not want to get swapped out. It cannot use
        // taskENTER_CRITICAL ()/taskEXIT_CRITICAL () as the length of the
        // operation may cause interrupts to be missed - including the
        // ticks.

        // Prevent the real time kernel swapping out the task.
        vTaskSuspendAll ();

        // Perform the operation here. There is no need to use critical
        // sections as we have all the microcontroller processing time.
        // During this time interrupts will still operate and the real
        // time kernel tick count will be maintained.
```

(continues on next page)

```

// ...

// The operation is complete. Restart the kernel. We want to force
// a context switch - but there is no point if resuming the scheduler
// caused a context switch already.
if( !xTaskResumeAll () )
{
    taskYIELD ();
}
}
}

```

Return If resuming the scheduler caused a context switch then pdTRUE is returned, otherwise pdFALSE is returned.

TickType_t **xTaskGetTickCount** (void)
Get tick count

Return The count of ticks since vTaskStartScheduler was called.

TickType_t **xTaskGetTickCountFromISR** (void)
Get tick count from ISR

This is a version of xTaskGetTickCount() that is safe to be called from an ISR - provided that TickType_t is the natural word size of the microcontroller being used or interrupt nesting is either not supported or not being used.

Return The count of ticks since vTaskStartScheduler was called.

UBaseType_t **uxTaskGetNumberOfTasks** (void)
Get current number of tasks

Return The number of tasks that the real time kernel is currently managing. This includes all ready, blocked and suspended tasks. A task that has been deleted but not yet freed by the idle task will also be included in the count.

char ***pcTaskGetTaskName** (*TaskHandle_t* xTaskToQuery)
Get task name

Return The text (human readable) name of the task referenced by the handle xTaskToQuery. A task can query its own name by either passing in its own handle, or by setting xTaskToQuery to NULL. INCLUDE_pcTaskGetTaskName must be set to 1 in FreeRTOSConfig.h for pcTaskGetTaskName() to be available.

UBaseType_t **uxTaskGetStackHighWaterMark** (*TaskHandle_t* xTask)
Returns the high water mark of the stack associated with xTask.

INCLUDE_uxTaskGetStackHighWaterMark must be set to 1 in FreeRTOSConfig.h for this function to be available.

High water mark is the minimum free stack space there has been (in bytes rather than words as found in vanilla FreeRTOS) since the task started. The smaller the returned number the closer the task has come to overflowing its stack.

Return The smallest amount of free stack space there has been (in bytes rather than words as found in vanilla FreeRTOS) since the task referenced by xTask was created.

Parameters

- xTask: Handle of the task associated with the stack to be checked. Set xTask to NULL to check the stack of the calling task.

uint8_t ***pxTaskGetStackStart** (*TaskHandle_t* xTask)
Returns the start of the stack associated with xTask.

INCLUDE_pxTaskGetStackStart must be set to 1 in FreeRTOSConfig.h for this function to be available.

Returns the highest stack memory address on architectures where the stack grows down from high memory, and the lowest memory address on architectures where the stack grows up from low memory.

Return A pointer to the start of the stack.

Parameters

- `xTask`: Handle of the task associated with the stack returned. Set `xTask` to `NULL` to return the stack of the calling task.

void **vTaskSetApplicationTaskTag** (*TaskHandle_t* `xTask`, *TaskHookFunction_t* `pxHookFunction`)
Sets `pxHookFunction` to be the task hook function used by the task `xTask`.

Parameters

- `xTask`: Handle of the task to set the hook function for. Passing `xTask` as `NULL` has the effect of setting the calling task's hook function.
- `pxHookFunction`: Pointer to the hook function.

TaskHookFunction_t **xTaskGetApplicationTaskTag** (*TaskHandle_t* `xTask`)

Get the hook function assigned to given task.

Return The `pxHookFunction` value assigned to the task `xTask`.

Parameters

- `xTask`: Handle of the task to get the hook function for. Passing `xTask` as `NULL` has the effect of getting the calling task's hook function.

void **vTaskSetThreadLocalStoragePointer** (*TaskHandle_t* `xTaskToSet`, *BaseType_t* `xIndex`, void **pvValue*)

Set local storage pointer specific to the given task.

Each task contains an array of pointers that is dimensioned by the `configNUM_THREAD_LOCAL_STORAGE_POINTERS` setting in `FreeRTOSConfig.h`. The kernel does not use the pointers itself, so the application writer can use the pointers for any purpose they wish.

Parameters

- `xTaskToSet`: Task to set thread local storage pointer for
- `xIndex`: The index of the pointer to set, from 0 to `configNUM_THREAD_LOCAL_STORAGE_POINTERS - 1`.
- `pvValue`: Pointer value to set.

void ***pvTaskGetThreadLocalStoragePointer** (*TaskHandle_t* `xTaskToQuery`, *BaseType_t* `xIndex`)

Get local storage pointer specific to the given task.

Each task contains an array of pointers that is dimensioned by the `configNUM_THREAD_LOCAL_STORAGE_POINTERS` setting in `FreeRTOSConfig.h`. The kernel does not use the pointers itself, so the application writer can use the pointers for any purpose they wish.

Return Pointer value

Parameters

- `xTaskToQuery`: Task to get thread local storage pointer for
- `xIndex`: The index of the pointer to get, from 0 to `configNUM_THREAD_LOCAL_STORAGE_POINTERS - 1`.

void **vTaskSetThreadLocalStoragePointerAndDelCallback** (*TaskHandle_t* `xTaskToSet`, *BaseType_t* `xIndex`, void **pvValue*, *TlsDeleteCallbackFunction_t* `pvDelCallback`)

Set local storage pointer and deletion callback.

Each task contains an array of pointers that is dimensioned by the `configNUM_THREAD_LOCAL_STORAGE_POINTERS` setting in `FreeRTOSConfig.h`. The kernel does not use the pointers itself, so the application writer can use the pointers for any purpose they wish.

Local storage pointers set for a task can reference dynamically allocated resources. This function is similar to `vTaskSetThreadLocalStoragePointer`, but provides a way to release these resources when the task gets deleted. For each pointer, a callback function can be set. This function will be called when task is deleted, with the local storage pointer index and value as arguments.

Parameters

- **xTaskToSet**: Task to set thread local storage pointer for
- **xIndex**: The index of the pointer to set, from 0 to `configNUM_THREAD_LOCAL_STORAGE_POINTERS - 1`.
- **pvValue**: Pointer value to set.
- **pvDelCallback**: Function to call to dispose of the local storage pointer when the task is deleted.

`BaseType_t` **xTaskCallApplicationTaskHook** (*TaskHandle_t* *xTask*, void **pvParameter*)

Calls the hook function associated with *xTask*. Passing *xTask* as NULL has the effect of calling the Running tasks (the calling task) hook function.

Parameters

- **xTask**: Handle of the task to call the hook for.
- **pvParameter**: Parameter passed to the hook function for the task to interpret as it wants. The return value is the value returned by the task hook function registered by the user.

TaskHandle_t **xTaskGetIdleTaskHandle** (void)

Get the handle of idle task for the current CPU.

`xTaskGetIdleTaskHandle()` is only available if `INCLUDE_xTaskGetIdleTaskHandle` is set to 1 in `FreeRTOSConfig.h`.

Return The handle of the idle task. It is not valid to call `xTaskGetIdleTaskHandle()` before the scheduler has been started.

TaskHandle_t **xTaskGetIdleTaskHandleForCPU** (`UBaseType_t` *cpuid*)

Get the handle of idle task for the given CPU.

`xTaskGetIdleTaskHandleForCPU()` is only available if `INCLUDE_xTaskGetIdleTaskHandle` is set to 1 in `FreeRTOSConfig.h`.

Return Idle task handle of a given cpu. It is not valid to call `xTaskGetIdleTaskHandleForCPU()` before the scheduler has been started.

Parameters

- **cpuid**: The CPU to get the handle for

`UBaseType_t` **uxTaskGetSystemState** (*TaskStatus_t* ***const** *pxTaskStatusArray*, **const** `UBaseType_t` *uxArraySize*, `uint32_t` ***const** *pulTotalRunTime*)

Get the state of tasks in the system.

`configUSE_TRACE_FACILITY` must be defined as 1 in `FreeRTOSConfig.h` for `uxTaskGetSystemState()` to be available.

`uxTaskGetSystemState()` populates an `TaskStatus_t` structure for each task in the system. `TaskStatus_t` structures contain, among other things, members for the task handle, task name, task priority, task state, and total amount of run time consumed by the task. See the `TaskStatus_t` structure definition in this file for the full member list.

Example usage:

```
// This example demonstrates how a human readable table of run time stats
// information is generated from raw data provided by uxTaskGetSystemState().
// The human readable table is written to pcWriteBuffer
void vTaskGetRunTimeStats( char *pcWriteBuffer )
{
    TaskStatus_t *pxTaskStatusArray;
    volatile UBaseType_t uxArraySize, x;
    uint32_t ulTotalRunTime, ulStatsAsPercentage;

    // Make sure the write buffer does not contain a string.
    *pcWriteBuffer = 0x00;

    // Take a snapshot of the number of tasks in case it changes while this
    // function is executing.
    uxArraySize = uxTaskGetNumberOfTasks();
```

(continues on next page)

(continued from previous page)

```

// Allocate a TaskStatus_t structure for each task. An array could be
// allocated statically at compile time.
pxTaskStatusArray = pvPortMalloc( uxArraySize * sizeof( TaskStatus_t ) );

if( pxTaskStatusArray != NULL )
{
    // Generate raw status information about each task.
    uxArraySize = uxTaskGetSystemState( pxTaskStatusArray, uxArraySize, &
↳ulTotalRunTime );

    // For percentage calculations.
    ulTotalRunTime /= 100UL;

    // Avoid divide by zero errors.
    if( ulTotalRunTime > 0 )
    {
        // For each populated position in the pxTaskStatusArray array,
        // format the raw data as human readable ASCII data
        for( x = 0; x < uxArraySize; x++ )
        {
            // What percentage of the total run time has the task used?
            // This will always be rounded down to the nearest integer.
            // ulTotalRunTimeDiv100 has already been divided by 100.
            ulStatsAsPercentage = pxTaskStatusArray[ x ].ulRunTimeCounter /↳
↳ulTotalRunTime;

            if( ulStatsAsPercentage > 0UL )
            {
                sprintf( pcWriteBuffer, "%s\t\t%lu\t\t%lu%%\r\n",↳
↳pxTaskStatusArray[ x ].pcTaskName, pxTaskStatusArray[ x ].ulRunTimeCounter,↳
↳ulStatsAsPercentage );
            }
            else
            {
                // If the percentage is zero here then the task has
                // consumed less than 1% of the total run time.
                sprintf( pcWriteBuffer, "%s\t\t%lu\t\t<1%%\r\n",↳
↳pxTaskStatusArray[ x ].pcTaskName, pxTaskStatusArray[ x ].ulRunTimeCounter );
            }

            pcWriteBuffer += strlen( ( char * ) pcWriteBuffer );
        }

        // The array is no longer needed, free the memory it consumes.
        vPortFree( pxTaskStatusArray );
    }
}

```

Note This function is intended for debugging use only as its use results in the scheduler remaining suspended for an extended period.

Return The number of TaskStatus_t structures that were populated by uxTaskGetSystemState(). This should equal the number returned by the uxTaskGetNumberOfTasks() API function, but will be zero if the value passed in the uxArraySize parameter was too small.

Parameters

- pxTaskStatusArray: A pointer to an array of TaskStatus_t structures. The array must contain at least one TaskStatus_t structure for each task that is under the control of the RTOS. The number of tasks under the control of the RTOS can be determined using the uxTaskGetNumberOfTasks() API function.
- uxArraySize: The size of the array pointed to by the pxTaskStatusArray parameter. The size is

specified as the number of indexes in the array, or the number of `TaskStatus_t` structures contained in the array, not by the number of bytes in the array.

- `pulTotalRunTime`: If `configGENERATE_RUN_TIME_STATS` is set to 1 in `FreeRTOSConfig.h` then `*pulTotalRunTime` is set by `uxTaskGetSystemState()` to the total run time (as defined by the run time stats clock, see <http://www.freertos.org/rtos-run-time-stats.html>) since the target booted. `pulTotalRunTime` can be set to `NULL` to omit the total run time information.

void **vTaskList** (char *pcWriteBuffer)

List all the current tasks.

`configUSE_TRACE_FACILITY` and `configUSE_STATS_FORMATTING_FUNCTIONS` must both be defined as 1 for this function to be available. See the configuration section of the FreeRTOS.org website for more information.

Lists all the current tasks, along with their current state and stack usage high water mark.

Note This function will disable interrupts for its duration. It is not intended for normal application runtime use but as a debug aid.

Tasks are reported as blocked ('B'), ready ('R'), deleted ('D') or suspended ('S').

`vTaskList()` calls `uxTaskGetSystemState()`, then formats part of the `uxTaskGetSystemState()` output into a human readable table that displays task names, states and stack usage.

Note This function is provided for convenience only, and is used by many of the demo applications. Do not consider it to be part of the scheduler.

`vTaskList()` has a dependency on the `sprintf()` C library function that might bloat the code size, use a lot of stack, and provide different results on different platforms. An alternative, tiny, third party, and limited functionality implementation of `sprintf()` is provided in many of the FreeRTOS/Demo sub-directories in a file called `printf-stdarg.c` (note `printf-stdarg.c` does not provide a full `snprintf()` implementation!).

It is recommended that production systems call `uxTaskGetSystemState()` directly to get access to raw stats data, rather than indirectly through a call to `vTaskList()`.

Parameters

- `pcWriteBuffer`: A buffer into which the above mentioned details will be written, in ASCII form. This buffer is assumed to be large enough to contain the generated report. Approximately 40 bytes per task should be sufficient.

void **vTaskGetRunTimeStats** (char *pcWriteBuffer)

Get the state of running tasks as a string

`configGENERATE_RUN_TIME_STATS` and `configUSE_STATS_FORMATTING_FUNCTIONS` must both be defined as 1 for this function to be available. The application must also then provide definitions for `portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()` and `portGET_RUN_TIME_COUNTER_VALUE()` to configure a peripheral timer/counter and return the timers current count value respectively. The counter should be at least 10 times the frequency of the tick count.

Setting `configGENERATE_RUN_TIME_STATS` to 1 will result in a total accumulated execution time being stored for each task. The resolution of the accumulated time value depends on the frequency of the timer configured by the `portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()` macro. Calling `vTaskGetRunTimeStats()` writes the total execution time of each task into a buffer, both as an absolute count value and as a percentage of the total system execution time.

Note This function will disable interrupts for its duration. It is not intended for normal application runtime use but as a debug aid.

`vTaskGetRunTimeStats()` calls `uxTaskGetSystemState()`, then formats part of the `uxTaskGetSystemState()` output into a human readable table that displays the amount of time each task has spent in the Running state in both absolute and percentage terms.

Note This function is provided for convenience only, and is used by many of the demo applications. Do not consider it to be part of the scheduler.

`vTaskGetRunTimeStats()` has a dependency on the `sprintf()` C library function that might bloat the code size, use a lot of stack, and provide different results on different platforms. An alternative, tiny, third party, and

limited functionality implementation of `sprintf()` is provided in many of the FreeRTOS/Demo sub-directories in a file called `printf-stdarg.c` (note `printf-stdarg.c` does not provide a full `snprintf()` implementation!).

It is recommended that production systems call `uxTaskGetSystemState()` directly to get access to raw stats data, rather than indirectly through a call to `vTaskGetRunTimeStats()`.

Parameters

- `pcWriteBuffer`: A buffer into which the execution times will be written, in ASCII form. This buffer is assumed to be large enough to contain the generated report. Approximately 40 bytes per task should be sufficient.

BaseType_t **xTaskNotify** (*TaskHandle_t xTaskToNotify*, uint32_t *ulValue*, *eNotifyAction eAction*)

Send task notification.

`configUSE_TASK_NOTIFICATIONS` must be undefined or defined as 1 for this function to be available.

When `configUSE_TASK_NOTIFICATIONS` is set to one each task has its own private “notification value”, which is a 32-bit unsigned integer (`uint32_t`).

Events can be sent to a task using an intermediary object. Examples of such objects are queues, semaphores, mutexes and event groups. Task notifications are a method of sending an event directly to a task without the need for such an intermediary object.

A notification sent to a task can optionally perform an action, such as update, overwrite or increment the task’s notification value. In that way task notifications can be used to send data to a task, or be used as light weight and fast binary or counting semaphores.

A notification sent to a task will remain pending until it is cleared by the task calling `xTaskNotifyWait()` or `ulTaskNotifyTake()`. If the task was already in the Blocked state to wait for a notification when the notification arrives then the task will automatically be removed from the Blocked state (unblocked) and the notification cleared.

A task can use `xTaskNotifyWait()` to [optionally] block to wait for a notification to be pending, or `ulTaskNotifyTake()` to [optionally] block to wait for its notification value to have a non-zero value. The task does not consume any CPU time while it is in the Blocked state.

See <http://www.FreeRTOS.org/RTOS-task-notifications.html> for details.

Return Dependent on the value of `eAction`. See the description of the `eAction` parameter.

Parameters

- `xTaskToNotify`: The handle of the task being notified. The handle to a task can be returned from the `xTaskCreate()` API function used to create the task, and the handle of the currently running task can be obtained by calling `xTaskGetCurrentTaskHandle()`.
- `ulValue`: Data that can be sent with the notification. How the data is used depends on the value of the `eAction` parameter.
- `eAction`: Specifies how the notification updates the task’s notification value, if at all. Valid values for `eAction` are as follows:
 - `eSetBits`: The task’s notification value is bitwise ORed with `ulValue`. `xTaskNotify()` always returns `pdPASS` in this case.
 - `eIncrement`: The task’s notification value is incremented. `ulValue` is not used and `xTaskNotify()` always returns `pdPASS` in this case.
 - `eSetValueWithOverwrite`: The task’s notification value is set to the value of `ulValue`, even if the task being notified had not yet processed the previous notification (the task already had a notification pending). `xTaskNotify()` always returns `pdPASS` in this case.
 - `eSetValueWithoutOverwrite`: If the task being notified did not already have a notification pending then the task’s notification value is set to `ulValue` and `xTaskNotify()` will return `pdPASS`. If the task being notified already had a notification pending then no action is performed and `pdFAIL` is returned.
 - `eNoAction`: The task receives a notification without its notification value being updated. `ulValue` is not used and `xTaskNotify()` always returns `pdPASS` in this case.

BaseType_t **xTaskNotifyFromISR** (*TaskHandle_t xTaskToNotify*, uint32_t *ulValue*, *eNotifyAction eAction*, BaseType_t **pxHigherPriorityTaskWoken*)

Send task notification from an ISR.

`configUSE_TASK_NOTIFICATIONS` must be undefined or defined as 1 for this function to be available.

When `configUSE_TASK_NOTIFICATIONS` is set to one each task has its own private “notification value”, which is a 32-bit unsigned integer (`uint32_t`).

A version of `xTaskNotify()` that can be used from an interrupt service routine (ISR).

Events can be sent to a task using an intermediary object. Examples of such objects are queues, semaphores, mutexes and event groups. Task notifications are a method of sending an event directly to a task without the need for such an intermediary object.

A notification sent to a task can optionally perform an action, such as update, overwrite or increment the task’s notification value. In that way task notifications can be used to send data to a task, or be used as light weight and fast binary or counting semaphores.

A notification sent to a task will remain pending until it is cleared by the task calling `xTaskNotifyWait()` or `ulTaskNotifyTake()`. If the task was already in the Blocked state to wait for a notification when the notification arrives then the task will automatically be removed from the Blocked state (unblocked) and the notification cleared.

A task can use `xTaskNotifyWait()` to [optionally] block to wait for a notification to be pending, or `ulTaskNotifyTake()` to [optionally] block to wait for its notification value to have a non-zero value. The task does not consume any CPU time while it is in the Blocked state.

See <http://www.FreeRTOS.org/RTOS-task-notifications.html> for details.

Return Dependent on the value of `eAction`. See the description of the `eAction` parameter.

Parameters

- `xTaskToNotify`: The handle of the task being notified. The handle to a task can be returned from the `xTaskCreate()` API function used to create the task, and the handle of the currently running task can be obtained by calling `xTaskGetCurrentTaskHandle()`.
- `ulValue`: Data that can be sent with the notification. How the data is used depends on the value of the `eAction` parameter.
- `eAction`: Specifies how the notification updates the task’s notification value, if at all. Valid values for `eAction` are as follows:
 - `eSetBits`: The task’s notification value is bitwise ORed with `ulValue`. `xTaskNotify()` always returns `pdPASS` in this case.
 - `eIncrement`: The task’s notification value is incremented. `ulValue` is not used and `xTaskNotify()` always returns `pdPASS` in this case.
 - `eSetValueWithOverwrite`: The task’s notification value is set to the value of `ulValue`, even if the task being notified had not yet processed the previous notification (the task already had a notification pending). `xTaskNotify()` always returns `pdPASS` in this case.
 - `eSetValueWithoutOverwrite`: If the task being notified did not already have a notification pending then the task’s notification value is set to `ulValue` and `xTaskNotify()` will return `pdPASS`. If the task being notified already had a notification pending then no action is performed and `pdFAIL` is returned.
 - `eNoAction`: The task receives a notification without its notification value being updated. `ulValue` is not used and `xTaskNotify()` always returns `pdPASS` in this case.
- `pxHigherPriorityTaskWoken`: `xTaskNotifyFromISR()` will set `*pxHigherPriorityTaskWoken` to `pdTRUE` if sending the notification caused the task to which the notification was sent to leave the Blocked state, and the unblocked task has a priority higher than the currently running task. If `xTaskNotifyFromISR()` sets this value to `pdTRUE` then a context switch should be requested before the interrupt is exited. How a context switch is requested from an ISR is dependent on the port - see the documentation page for the port in use.

`BaseType_t xTaskNotifyWait` (`uint32_t ulBitsToClearOnEntry`, `uint32_t ulBitsToClearOnExit`, `uint32_t *pulNotificationValue`, `TickType_t xTicksToWait`)

Wait for task notification

`configUSE_TASK_NOTIFICATIONS` must be undefined or defined as 1 for this function to be available.

When `configUSE_TASK_NOTIFICATIONS` is set to one each task has its own private “notification value”, which is a 32-bit unsigned integer (`uint32_t`).

Events can be sent to a task using an intermediary object. Examples of such objects are queues, semaphores, mutexes and event groups. Task notifications are a method of sending an event directly to a task without the need for such an intermediary object.

A notification sent to a task can optionally perform an action, such as update, overwrite or increment the task's notification value. In that way task notifications can be used to send data to a task, or be used as light weight and fast binary or counting semaphores.

A notification sent to a task will remain pending until it is cleared by the task calling `xTaskNotifyWait()` or `ulTaskNotifyTake()`. If the task was already in the Blocked state to wait for a notification when the notification arrives then the task will automatically be removed from the Blocked state (unblocked) and the notification cleared.

A task can use `xTaskNotifyWait()` to [optionally] block to wait for a notification to be pending, or `ulTaskNotifyTake()` to [optionally] block to wait for its notification value to have a non-zero value. The task does not consume any CPU time while it is in the Blocked state.

See <http://www.FreeRTOS.org/RTOS-task-notifications.html> for details.

Return If a notification was received (including notifications that were already pending when `xTaskNotifyWait` was called) then `pdPASS` is returned. Otherwise `pdFAIL` is returned.

Parameters

- `ulBitsToClearOnEntry`: Bits that are set in `ulBitsToClearOnEntry` value will be cleared in the calling task's notification value before the task checks to see if any notifications are pending, and optionally blocks if no notifications are pending. Setting `ulBitsToClearOnEntry` to `ULONG_MAX` (if `limits.h` is included) or `0xffffffffUL` (if `limits.h` is not included) will have the effect of resetting the task's notification value to 0. Setting `ulBitsToClearOnEntry` to 0 will leave the task's notification value unchanged.
- `ulBitsToClearOnExit`: If a notification is pending or received before the calling task exits the `xTaskNotifyWait()` function then the task's notification value (see the `xTaskNotify()` API function) is passed out using the `pulNotificationValue` parameter. Then any bits that are set in `ulBitsToClearOnExit` will be cleared in the task's notification value (note `*pulNotificationValue` is set before any bits are cleared). Setting `ulBitsToClearOnExit` to `ULONG_MAX` (if `limits.h` is included) or `0xffffffffUL` (if `limits.h` is not included) will have the effect of resetting the task's notification value to 0 before the function exits. Setting `ulBitsToClearOnExit` to 0 will leave the task's notification value unchanged when the function exits (in which case the value passed out in `pulNotificationValue` will match the task's notification value).
- `pulNotificationValue`: Used to pass the task's notification value out of the function. Note the value passed out will not be effected by the clearing of any bits caused by `ulBitsToClearOnExit` being non-zero.
- `xTicksToWait`: The maximum amount of time that the task should wait in the Blocked state for a notification to be received, should a notification not already be pending when `xTaskNotifyWait()` was called. The task will not consume any processing time while it is in the Blocked state. This is specified in kernel ticks, the macro `pdMS_TO_TICSK(value_in_ms)` can be used to convert a time specified in milliseconds to a time specified in ticks.

```
void vTaskNotifyGiveFromISR (TaskHandle_t xTaskToNotify, BaseType_t
                             *pxHigherPriorityTaskWoken)
```

Simplified macro for sending task notification from ISR.

`configUSE_TASK_NOTIFICATIONS` must be undefined or defined as 1 for this macro to be available.

When `configUSE_TASK_NOTIFICATIONS` is set to one each task has its own private "notification value", which is a 32-bit unsigned integer (`uint32_t`).

A version of `xTaskNotifyGive()` that can be called from an interrupt service routine (ISR).

Events can be sent to a task using an intermediary object. Examples of such objects are queues, semaphores, mutexes and event groups. Task notifications are a method of sending an event directly to a task without the need for such an intermediary object.

A notification sent to a task can optionally perform an action, such as update, overwrite or increment the task's notification value. In that way task notifications can be used to send data to a task, or be used as light weight and fast binary or counting semaphores.

`vTaskNotifyGiveFromISR()` is intended for use when task notifications are used as light weight and faster binary or counting semaphore equivalents. Actual FreeRTOS semaphores are given from an ISR using the `xSemaphoreGiveFromISR()` API function, the equivalent action that instead uses a task notification is `vTaskNotifyGiveFromISR()`.

When task notifications are being used as a binary or counting semaphore equivalent then the task being notified should wait for the notification using the `ulTaskNotificationTake()` API function rather than the `xTaskNotifyWait()` API function.

See <http://www.FreeRTOS.org/RTOS-task-notifications.html> for more details.

Parameters

- `xTaskToNotify`: The handle of the task being notified. The handle to a task can be returned from the `xTaskCreate()` API function used to create the task, and the handle of the currently running task can be obtained by calling `xTaskGetCurrentTaskHandle()`.
- `pxHigherPriorityTaskWoken`: `vTaskNotifyGiveFromISR()` will set `*pxHigherPriorityTaskWoken` to `pdTRUE` if sending the notification caused the task to which the notification was sent to leave the Blocked state, and the unblocked task has a priority higher than the currently running task. If `vTaskNotifyGiveFromISR()` sets this value to `pdTRUE` then a context switch should be requested before the interrupt is exited. How a context switch is requested from an ISR is dependent on the port - see the documentation page for the port in use.

`uint32_t ulTaskNotifyTake` (`BaseType_t xClearCountOnExit`, `TickType_t xTicksToWait`)

Simplified macro for receiving task notification.

`configUSE_TASK_NOTIFICATIONS` must be undefined or defined as 1 for this function to be available.

When `configUSE_TASK_NOTIFICATIONS` is set to one each task has its own private “notification value”, which is a 32-bit unsigned integer (`uint32_t`).

Events can be sent to a task using an intermediary object. Examples of such objects are queues, semaphores, mutexes and event groups. Task notifications are a method of sending an event directly to a task without the need for such an intermediary object.

A notification sent to a task can optionally perform an action, such as update, overwrite or increment the task’s notification value. In that way task notifications can be used to send data to a task, or be used as light weight and fast binary or counting semaphores.

`ulTaskNotifyTake()` is intended for use when a task notification is used as a faster and lighter weight binary or counting semaphore alternative. Actual FreeRTOS semaphores are taken using the `xSemaphoreTake()` API function, the equivalent action that instead uses a task notification is `ulTaskNotifyTake()`.

When a task is using its notification value as a binary or counting semaphore other tasks should send notifications to it using the `xTaskNotifyGive()` macro, or `xTaskNotify()` function with the `eAction` parameter set to `eIncrement`.

`ulTaskNotifyTake()` can either clear the task’s notification value to zero on exit, in which case the notification value acts like a binary semaphore, or decrement the task’s notification value on exit, in which case the notification value acts like a counting semaphore.

A task can use `ulTaskNotifyTake()` to [optionally] block to wait for a the task’s notification value to be non-zero. The task does not consume any CPU time while it is in the Blocked state.

Where as `xTaskNotifyWait()` will return when a notification is pending, `ulTaskNotifyTake()` will return when the task’s notification value is not zero.

See <http://www.FreeRTOS.org/RTOS-task-notifications.html> for details.

Return The task’s notification count before it is either cleared to zero or decremented (see the `xClearCountOnExit` parameter).

Parameters

- `xClearCountOnExit`: if `xClearCountOnExit` is `pdFALSE` then the task’s notification value is decremented when the function exits. In this way the notification value acts like a counting semaphore. If `xClearCountOnExit` is not `pdFALSE` then the task’s notification value is cleared to zero when the function exits. In this way the notification value acts like a binary semaphore.

- `xTicksToWait`: The maximum amount of time that the task should wait in the Blocked state for the task's notification value to be greater than zero, should the count not already be greater than zero when `ulTaskNotifyTake()` was called. The task will not consume any processing time while it is in the Blocked state. This is specified in kernel ticks, the macro `pdMS_TO_TICSK(value_in_ms)` can be used to convert a time specified in milliseconds to a time specified in ticks.

Structures

struct xTASK_STATUS

Used with the `uxTaskGetSystemState()` function to return the state of each task in the system.

Public Members

TaskHandle_t **xHandle**

The handle of the task to which the rest of the information in the structure relates.

const char *pcTaskName

A pointer to the task's name. This value will be invalid if the task was deleted since the structure was populated!

UBaseType_t **xTaskNumber**

A number unique to the task.

eTaskState **eCurrentState**

The state in which the task existed when the structure was populated.

UBaseType_t **uxCurrentPriority**

The priority at which the task was running (may be inherited) when the structure was populated.

UBaseType_t **uxBasePriority**

The priority to which the task will return if the task's current priority has been inherited to avoid unbounded priority inversion when obtaining a mutex. Only valid if `configUSE_MUTEXES` is defined as 1 in `FreeRTOSConfig.h`.

uint32_t **ulRunTimeCounter**

The total run time allocated to the task so far, as defined by the run time stats clock. See <http://www.freertos.org/rtos-run-time-stats.html>. Only valid when `configGENERATE_RUN_TIME_STATS` is defined as 1 in `FreeRTOSConfig.h`.

StackType_t ***pxStackBase**

Points to the lowest address of the task's stack area.

uint32_t **usStackHighWaterMark**

The minimum amount of stack space that has remained for the task since the task was created. The closer this value is to zero the closer the task has come to overflowing its stack.

BaseType_t **xCoreID**

Core this task is pinned to (0, 1, or -1 for `tskNO_AFFINITY`). This field is present if `CONFIG_FREERTOS_VTASKLIST_INCLUDE_COREID` is set.

struct xTASK_SNAPSHOT

Used with the `uxTaskGetSnapshotAll()` function to save memory snapshot of each task in the system. We need this struct because `TCB_t` is defined (hidden) in `tasks.c`.

Public Members

void ***pxTCB**

Address of task control block.

StackType_t ***pxTopOfStack**

Points to the location of the last item placed on the tasks stack.

StackType_t ***pxEndOfStack**

Points to the end of the stack. `pxTopOfStack < pxEndOfStack`, stack grows hi2lo `pxTopOfStack > pxEndOfStack`, stack grows lo2hi

Macros

tskKERNEL_VERSION_NUMBER

tskKERNEL_VERSION_MAJOR

tskKERNEL_VERSION_MINOR

tskKERNEL_VERSION_BUILD

tskNO_AFFINITY

Argument of `xTaskCreatePinnedToCore` indicating that task has no affinity.

tskIDLE_PRIORITY

Defines the priority used by the idle task. This must not be modified.

taskYIELD ()

task. h

Macro for forcing a context switch.

taskENTER_CRITICAL (mux)

task. h

Macro to mark the start of a critical code region. Preemptive context switches cannot occur when in a critical region.

Note This may alter the stack (depending on the portable implementation) so must be used with care!

taskENTER_CRITICAL_ISR (mux)

taskEXIT_CRITICAL (mux)

task. h

Macro to mark the end of a critical code region. Preemptive context switches cannot occur when in a critical region.

Note This may alter the stack (depending on the portable implementation) so must be used with care!

taskEXIT_CRITICAL_ISR (mux)

taskDISABLE_INTERRUPTS ()

task. h

Macro to disable all maskable interrupts.

taskENABLE_INTERRUPTS ()

task. h

Macro to enable microcontroller interrupts.

taskSCHEDULER_SUSPENDED

taskSCHEDULER_NOT_STARTED

taskSCHEDULER_RUNNING

xTaskNotifyGive (xTaskToNotify)

Simplified macro for sending task notification.

`configUSE_TASK_NOTIFICATIONS` must be undefined or defined as 1 for this macro to be available.

When `configUSE_TASK_NOTIFICATIONS` is set to one each task has its own private “notification value”, which is a 32-bit unsigned integer (`uint32_t`).

Events can be sent to a task using an intermediary object. Examples of such objects are queues, semaphores, mutexes and event groups. Task notifications are a method of sending an event directly to a task without the need for such an intermediary object.

A notification sent to a task can optionally perform an action, such as update, overwrite or increment the task's notification value. In that way task notifications can be used to send data to a task, or be used as light weight and fast binary or counting semaphores.

`xTaskNotifyGive()` is a helper macro intended for use when task notifications are used as light weight and faster binary or counting semaphore equivalents. Actual FreeRTOS semaphores are given using the `xSemaphoreGive()` API function, the equivalent action that instead uses a task notification is `xTaskNotifyGive()`.

When task notifications are being used as a binary or counting semaphore equivalent then the task being notified should wait for the notification using the `ulTaskNotificationTake()` API function rather than the `xTaskNotifyWait()` API function.

See <http://www.FreeRTOS.org/RTOS-task-notifications.html> for more details.

Return `xTaskNotifyGive()` is a macro that calls `xTaskNotify()` with the `eAction` parameter set to `eIncrement` - so `pdPASS` is always returned.

Parameters

- `xTaskToNotify`: The handle of the task being notified. The handle to a task can be returned from the `xTaskCreate()` API function used to create the task, and the handle of the currently running task can be obtained by calling `xTaskGetCurrentTaskHandle()`.

Type Definitions

```
typedef void *TaskHandle_t
task. h
```

Type by which tasks are referenced. For example, a call to `xTaskCreate` returns (via a pointer parameter) an `TaskHandle_t` variable that can then be used as a parameter to `vTaskDelete` to delete the task.

```
typedef BaseType_t (*TaskHookFunction_t) (void *)
```

Defines the prototype to which the application task hook function must conform.

```
typedef struct xTASK_STATUS TaskStatus_t
```

Used with the `uxTaskGetSystemState()` function to return the state of each task in the system.

```
typedef struct xTASK_SNAPSHOT TaskSnapshot_t
```

Used with the `uxTaskGetSnapshotAll()` function to save memory snapshot of each task in the system. We need this struct because `TCB_t` is defined (hidden) in `tasks.c`.

```
typedef void (*TlsDeleteCallbackFunction_t) (int, void *)
```

Prototype of local storage pointer deletion callback.

Enumerations

```
enum eTaskState
```

Task states returned by `eTaskGetState`.

Values:

```
eRunning = 0
```

A task is querying the state of itself, so must be running.

```
eReady
```

The task being queried is in a read or pending ready list.

```
eBlocked
```

The task being queried is in the Blocked state.

```
eSuspended
```

The task being queried is in the Suspended state, or is in the Blocked state with an infinite time out.

```
eDeleted
```

The task being queried has been deleted, but its TCB has not yet been freed.

```
enum eNotifyAction
```

Actions that can be performed when `vTaskNotify()` is called.

Values:

eNoAction = 0

Notify the task without updating its notify value.

eSetBits

Set bits in the task's notification value.

eIncrement

Increment the task's notification value.

eSetValueWithOverwrite

Set the task's notification value to a specific value even if the previous value has not yet been read by the task.

eSetValueWithoutOverwrite

Set the task's notification value if the previous value has been read by the task.

enum eSleepModeStatus

Possible return values for eTaskConfirmSleepModeStatus().

Values:

eAbortSleep = 0

A task has been made ready or a context switch pending since portSUPPRESS_TICKS_AND_SLEEP() was called - abort entering a sleep mode.

eStandardSleep

Enter a sleep mode that will not last any longer than the expected idle time.

eNoTasksWaitingTimeout

No tasks are waiting for a timeout so it is safe to enter a sleep mode that can only be exited by an external interrupt.

Queue API**Header File**

- [freertos/include/freertos/queue.h](#)

Functions

BaseType_t **xQueueGenericSendFromISR** (*QueueHandle_t xQueue*, **const** void ***const** *pvItemToQueue*, BaseType_t ***const** *pxHigherPriorityTaskWoken*, **const** BaseType_t *xCopyPosition*)

It is preferred that the macros xQueueSendFromISR(), xQueueSendToFrontFromISR() and xQueueSendToBackFromISR() be used in place of calling this function directly. xQueueGiveFromISR() is an equivalent for use by semaphores that don't actually copy any data.

Post an item on a queue. It is safe to use this function from within an interrupt service routine.

Items are queued by copy not reference so it is preferable to only queue small items, especially when called from an ISR. In most cases it would be preferable to store a pointer to the item being queued.

Example usage for buffered IO (where the ISR can obtain more than one value per call):

```
void vBufferISR( void )
{
char cIn;
BaseType_t xHigherPriorityTaskWokenByPost;

// We have not woken a task at the start of the ISR.
xHigherPriorityTaskWokenByPost = pdFALSE;

// Loop until the buffer is empty.
do
{
```

(continues on next page)

(continued from previous page)

```

// Obtain a byte from the buffer.
cIn = portINPUT_BYTE( RX_REGISTER_ADDRESS );

// Post each byte.
xQueueGenericSendFromISR( xRxQueue, &cIn, &
→xHigherPriorityTaskWokenByPost, queueSEND_TO_BACK );

} while( portINPUT_BYTE( BUFFER_COUNT ) );

// Now the buffer is empty we can switch context if necessary. Note that_
→the
// name of the yield function required is port specific.
if( xHigherPriorityTaskWokenByPost )
{
    taskYIELD_YIELD_FROM_ISR();
}
}

```

Return pdTRUE if the data was successfully sent to the queue, otherwise errQUEUE_FULL.

Parameters

- xQueue: The handle to the queue on which the item is to be posted.
- pvItemToQueue: A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
- [out] pxHigherPriorityTaskWoken: xQueueGenericSendFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE if sending to the queue caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If xQueueGenericSendFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited.
- xCopyPosition: Can take the value queueSEND_TO_BACK to place the item at the back of the queue, or queueSEND_TO_FRONT to place the item at the front of the queue (for high priority messages).

BaseType_t **xQueueGiveFromISR** (*QueueHandle_t* xQueue, BaseType_t ***const** pxHigherPriorityTaskWoken)

BaseType_t **xQueueIsQueueEmptyFromISR** (**const** *QueueHandle_t* xQueue)

Utilities to query queues that are safe to use from an ISR. These utilities should be used only from within an ISR, or within a critical section.

BaseType_t **xQueueIsQueueFullFromISR** (**const** *QueueHandle_t* xQueue)

UBaseType_t **uxQueueMessagesWaitingFromISR** (**const** *QueueHandle_t* xQueue)

BaseType_t **xQueueGenericSend** (*QueueHandle_t* xQueue, **const** void ***const** pvItemToQueue, TickType_t xTicksToWait, **const** BaseType_t xCopyPosition)

It is preferred that the macros xQueueSend(), xQueueSendToFront() and xQueueSendToBack() are used in place of calling this function directly.

Post an item on a queue. The item is queued by copy, not by reference. This function must not be called from an interrupt service routine. See xQueueSendFromISR () for an alternative which may be used in an ISR.

Example usage:

```

struct AMessage
{
    char ucMessageID;
    char ucData[ 20 ];
} xMessage;

uint32_t ulVar = 10UL;

void vATask( void *pvParameters )
{

```

(continues on next page)

(continued from previous page)

```

QueueHandle_t xQueue1, xQueue2;
struct AMessage *pxMessage;

// Create a queue capable of containing 10 uint32_t values.
xQueue1 = xQueueCreate( 10, sizeof( uint32_t ) );

// Create a queue capable of containing 10 pointers to AMessage structures.
// These should be passed by pointer as they contain a lot of data.
xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );

// ...

if( xQueue1 != 0 )
{
    // Send an uint32_t. Wait for 10 ticks for space to become
    // available if necessary.
    if( xQueueGenericSend( xQueue1, ( void * ) &ulVar, ( TickType_t ) 10, queueSEND_TO_BACK ) != pdPASS )
    {
        // Failed to post the message, even after 10 ticks.
    }
}

if( xQueue2 != 0 )
{
    // Send a pointer to a struct AMessage object. Don't block if the
    // queue is already full.
    pxMessage = & xMessage;
    xQueueGenericSend( xQueue2, ( void * ) &pxMessage, ( TickType_t ) 0, queueSEND_TO_BACK );
}

// ... Rest of task code.
}

```

Return pdTRUE if the item was successfully posted, otherwise errQUEUE_FULL.

Parameters

- **xQueue**: The handle to the queue on which the item is to be posted.
- **pvItemToQueue**: A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from **pvItemToQueue** into the queue storage area.
- **xTicksToWait**: The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0 and the queue is full. The time is defined in tick periods so the constant portTICK_PERIOD_MS should be used to convert to real time if this is required.
- **xCopyPosition**: Can take the value **queueSEND_TO_BACK** to place the item at the back of the queue, or **queueSEND_TO_FRONT** to place the item at the front of the queue (for high priority messages).

BaseType_t **xQueuePeekFromISR** (*QueueHandle_t xQueue*, void ***const** *pvBuffer*)

A version of **xQueuePeek()** that can be called from an interrupt service routine (ISR).

Receive an item from a queue without removing the item from the queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created.

Successfully received items remain on the queue so will be returned again by the next call, or a call to **xQueueReceive()**.

Return pdTRUE if an item was successfully received from the queue, otherwise pdFALSE.

Parameters

- **xQueue**: The handle to the queue from which the item is to be received.

- `pvBuffer`: Pointer to the buffer into which the received item will be copied.

BaseType_t **xQueueGenericReceive** (*QueueHandle_t* xQueue, void ***const** pvBuffer, TickType_t xTicksToWait, **const** BaseType_t xJustPeek)

It is preferred that the macro `xQueueReceive()` be used rather than calling this function directly.

Receive an item from a queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created.

This function must not be used in an interrupt service routine. See `xQueueReceiveFromISR` for an alternative that can.

Example usage:

```

struct AMessage
{
    char ucMessageID;
    char ucData[ 20 ];
} xMessage;

QueueHandle_t xQueue;

// Task to create a queue and post a value.
void vATask( void *pvParameters )
{
    struct AMessage *pxMessage;

    // Create a queue capable of containing 10 pointers to AMessage structures.
    // These should be passed by pointer as they contain a lot of data.
    xQueue = xQueueCreate( 10, sizeof( struct AMessage * ) );
    if( xQueue == 0 )
    {
        // Failed to create the queue.
    }

    // ...

    // Send a pointer to a struct AMessage object. Don't block if the
    // queue is already full.
    pxMessage = &xMessage;
    xQueueSend( xQueue, ( void * ) &pxMessage, ( TickType_t ) 0 );

    // ... Rest of task code.
}

// Task to receive from the queue.
void vADifferentTask( void *pvParameters )
{
    struct AMessage *pRxedMessage;

    if( xQueue != 0 )
    {
        // Receive a message on the created queue. Block for 10 ticks if a
        // message is not immediately available.
        if( xQueueGenericReceive( xQueue, &( pRxedMessage ), ( TickType_t ) 10,
→ ) )
        {
            // pRxedMessage now points to the struct AMessage variable posted
            // by vATask.
        }
    }

    // ... Rest of task code.
}

```

Return pdTRUE if an item was successfully received from the queue, otherwise pdFALSE.

Parameters

- `xQueue`: The handle to the queue from which the item is to be received.
- `pvBuffer`: Pointer to the buffer into which the received item will be copied.
- `xTicksToWait`: The maximum amount of time the task should block waiting for an item to receive should the queue be empty at the time of the call. The time is defined in tick periods so the constant `portTICK_PERIOD_MS` should be used to convert to real time if this is required. `xQueueGenericReceive()` will return immediately if the queue is empty and `xTicksToWait` is 0.
- `xJustPeek`: When set to true, the item received from the queue is not actually removed from the queue - meaning a subsequent call to `xQueueReceive()` will return the same item. When set to false, the item being received from the queue is also removed from the queue.

UBaseType_t **uxQueueMessagesWaiting** (const *QueueHandle_t* xQueue)

Return the number of messages stored in a queue.

Return The number of messages available in the queue.

Parameters

- `xQueue`: A handle to the queue being queried.

UBaseType_t **uxQueueSpacesAvailable** (const *QueueHandle_t* xQueue)

Return the number of free spaces available in a queue. This is equal to the number of items that can be sent to the queue before the queue becomes full if no items are removed.

Return The number of spaces available in the queue.

Parameters

- `xQueue`: A handle to the queue being queried.

void **vQueueDelete** (*QueueHandle_t* xQueue)

Delete a queue - freeing all the memory allocated for storing of items placed on the queue.

Parameters

- `xQueue`: A handle to the queue to be deleted.

BaseType_t **xQueueReceiveFromISR** (*QueueHandle_t* xQueue, void *const pvBuffer, BaseType_t *const pxHigherPriorityTaskWoken)

Receive an item from a queue. It is safe to use this function from within an interrupt service routine.

Example usage:

```
QueueHandle_t xQueue;

// Function to create a queue and post some values.
void vAFunction( void *pvParameters )
{
    char cValueToPost;
    const TickType_t xTicksToWait = ( TickType_t )0xffff;

    // Create a queue capable of containing 10 characters.
    xQueue = xQueueCreate( 10, sizeof( char ) );
    if( xQueue == 0 )
    {
        // Failed to create the queue.
    }

    // ...

    // Post some characters that will be used within an ISR. If the queue
    // is full then this task will block for xTicksToWait ticks.
    cValueToPost = 'a';
    xQueueSend( xQueue, ( void * ) &cValueToPost, xTicksToWait );
    cValueToPost = 'b';
    xQueueSend( xQueue, ( void * ) &cValueToPost, xTicksToWait );

    // ... keep posting characters ... this task may block when the queue
```

(continues on next page)

```

// becomes full.

cValueToPost = 'c';
xQueueSend( xQueue, ( void * ) &cValueToPost, xTicksToWait );
}

// ISR that outputs all the characters received on the queue.
void vISR_Routine( void )
{
BaseType_t xTaskWokenByReceive = pdFALSE;
char cRxdChar;

while( xQueueReceiveFromISR( xQueue, ( void * ) &cRxdChar, &
↪xTaskWokenByReceive) )
{
// A character was received. Output the character now.
vOutputCharacter( cRxdChar );

// If removing the character from the queue woke the task that was
// posting onto the queue cTaskWokenByReceive will have been set to
// pdTRUE. No matter how many times this loop iterates only one
// task will be woken.
}

if( cTaskWokenByPost != ( char ) pdFALSE;
{
taskYIELD ();
}
}
}

```

Return pdTRUE if an item was successfully received from the queue, otherwise pdFALSE.

Parameters

- xQueue: The handle to the queue from which the item is to be received.
- pvBuffer: Pointer to the buffer into which the received item will be copied.
- [out] pxHigherPriorityTaskWoken: A task may be blocked waiting for space to become available on the queue. If xQueueReceiveFromISR causes such a task to unblock *pxTaskWoken will get set to pdTRUE, otherwise *pxTaskWoken will remain unchanged.

void **vQueueAddToRegistry** (*QueueHandle_t* xQueue, const char *pcName)

The registry is provided as a means for kernel aware debuggers to locate queues, semaphores and mutexes. Call vQueueAddToRegistry() add a queue, semaphore or mutex handle to the registry if you want the handle to be available to a kernel aware debugger. If you are not using a kernel aware debugger then this function can be ignored.

configQUEUE_REGISTRY_SIZE defines the maximum number of handles the registry can hold. configQUEUE_REGISTRY_SIZE must be greater than 0 within FreeRTOSConfig.h for the registry to be available. Its value does not effect the number of queues, semaphores and mutexes that can be created - just the number that the registry can hold.

Parameters

- xQueue: The handle of the queue being added to the registry. This is the handle returned by a call to xQueueCreate(). Semaphore and mutex handles can also be passed in here.
- pcName: The name to be associated with the handle. This is the name that the kernel aware debugger will display. The queue registry only stores a pointer to the string - so the string must be persistent (global or preferably in ROM/Flash), not on the stack.

void **vQueueUnregisterQueue** (*QueueHandle_t* xQueue)

The registry is provided as a means for kernel aware debuggers to locate queues, semaphores and mutexes. Call vQueueAddToRegistry() add a queue, semaphore or mutex handle to the registry if you want the handle to be available to a kernel aware debugger, and vQueueUnregisterQueue() to remove the queue, semaphore or mutex from the register. If you are not using a kernel aware debugger then this function can be ignored.

Parameters

- `xQueue`: The handle of the queue being removed from the registry.

const char ***pcQueueGetName** (*QueueHandle_t* `xQueue`)

The queue registry is provided as a means for kernel aware debuggers to locate queues, semaphores and mutexes. Call `pcQueueGetName()` to look up and return the name of a queue in the queue registry from the queue's handle.

Note This function has been back ported from FreeRTOS v9.0.0

Return If the queue is in the registry then a pointer to the name of the queue is returned. If the queue is not in the registry then NULL is returned.

Parameters

- `xQueue`: The handle of the queue the name of which will be returned.

QueueHandle_t **xQueueGenericCreate** (**const** UBaseType_t `uxQueueLength`, **const** UBaseType_t `uxItemSize`, **const** uint8_t `ucQueueType`)

Generic version of the function used to create a queue using dynamic memory allocation. This is called by other functions and macros that create other RTOS objects that use the queue structure as their base.

QueueHandle_t **xQueueGenericCreateStatic** (**const** UBaseType_t `uxQueueLength`, **const** UBaseType_t `uxItemSize`, uint8_t `*pucQueueStorage`, StaticQueue_t `*pxStaticQueue`, **const** uint8_t `ucQueueType`)

Generic version of the function used to create a queue using dynamic memory allocation. This is called by other functions and macros that create other RTOS objects that use the queue structure as their base.

QueueSetHandle_t **xQueueCreateSet** (**const** UBaseType_t `uxEventQueueLength`)

Queue sets provide a mechanism to allow a task to block (pend) on a read operation from multiple queues or semaphores simultaneously.

See `FreeRTOS/Source/Demo/Common/Minimal/QueueSet.c` for an example using this function.

A queue set must be explicitly created using a call to `xQueueCreateSet()` before it can be used. Once created, standard FreeRTOS queues and semaphores can be added to the set using calls to `xQueueAddToSet()`. `xQueueSelectFromSet()` is then used to determine which, if any, of the queues or semaphores contained in the set is in a state where a queue read or semaphore take operation would be successful.

Note 1: See the documentation on <http://www.FreeRTOS.org/RTOS-queue-sets.html> for reasons why queue sets are very rarely needed in practice as there are simpler methods of blocking on multiple objects.

Note 2: Blocking on a queue set that contains a mutex will not cause the mutex holder to inherit the priority of the blocked task.

Note 3: An additional 4 bytes of RAM is required for each space in a every queue added to a queue set. Therefore counting semaphores that have a high maximum count value should not be added to a queue set.

Note 4: A receive (in the case of a queue) or take (in the case of a semaphore) operation must not be performed on a member of a queue set unless a call to `xQueueSelectFromSet()` has first returned a handle to that set member.

Return If the queue set is created successfully then a handle to the created queue set is returned. Otherwise NULL is returned.

Parameters

- `uxEventQueueLength`: Queue sets store events that occur on the queues and semaphores contained in the set. `uxEventQueueLength` specifies the maximum number of events that can be queued at once. To be absolutely certain that events are not lost `uxEventQueueLength` should be set to the total sum of the length of the queues added to the set, where binary semaphores and mutexes have a length of 1, and counting semaphores have a length set by their maximum count value. Examples:
 - If a queue set is to hold a queue of length 5, another queue of length 12, and a binary semaphore, then `uxEventQueueLength` should be set to $(5 + 12 + 1)$, or 18.
 - If a queue set is to hold three binary semaphores then `uxEventQueueLength` should be set to $(1 + 1 + 1)$, or 3.

- If a queue set is to hold a counting semaphore that has a maximum count of 5, and a counting semaphore that has a maximum count of 3, then `uxEventQueueLength` should be set to $(5 + 3)$, or 8.

BaseType_t **xQueueAddToSet** (*QueueSetMemberHandle_t* xQueueOrSemaphore, *QueueSetHandle_t* xQueueSet)

Adds a queue or semaphore to a queue set that was previously created by a call to `xQueueCreateSet()`.

See `FreeRTOS/Source/Demo/Common/Minimal/QueueSet.c` for an example using this function.

Note 1: A receive (in the case of a queue) or take (in the case of a semaphore) operation must not be performed on a member of a queue set unless a call to `xQueueSelectFromSet()` has first returned a handle to that set member.

Return If the queue or semaphore was successfully added to the queue set then `pdPASS` is returned. If the queue could not be successfully added to the queue set because it is already a member of a different queue set then `pdFAIL` is returned.

Parameters

- `xQueueOrSemaphore`: The handle of the queue or semaphore being added to the queue set (cast to an `QueueSetMemberHandle_t` type).
- `xQueueSet`: The handle of the queue set to which the queue or semaphore is being added.

BaseType_t **xQueueRemoveFromSet** (*QueueSetMemberHandle_t* xQueueOrSemaphore, *QueueSetHandle_t* xQueueSet)

Removes a queue or semaphore from a queue set. A queue or semaphore can only be removed from a set if the queue or semaphore is empty.

See `FreeRTOS/Source/Demo/Common/Minimal/QueueSet.c` for an example using this function.

Return If the queue or semaphore was successfully removed from the queue set then `pdPASS` is returned. If the queue was not in the queue set, or the queue (or semaphore) was not empty, then `pdFAIL` is returned.

Parameters

- `xQueueOrSemaphore`: The handle of the queue or semaphore being removed from the queue set (cast to an `QueueSetMemberHandle_t` type).
- `xQueueSet`: The handle of the queue set in which the queue or semaphore is included.

QueueSetMemberHandle_t **xQueueSelectFromSet** (*QueueSetHandle_t* xQueueSet, **const** TickType_t xTicksToWait)

`xQueueSelectFromSet()` selects from the members of a queue set a queue or semaphore that either contains data (in the case of a queue) or is available to take (in the case of a semaphore). `xQueueSelectFromSet()` effectively allows a task to block (pend) on a read operation on all the queues and semaphores in a queue set simultaneously.

See `FreeRTOS/Source/Demo/Common/Minimal/QueueSet.c` for an example using this function.

Note 1: See the documentation on <http://www.FreeRTOS.org/RTOS-queue-sets.html> for reasons why queue sets are very rarely needed in practice as there are simpler methods of blocking on multiple objects.

Note 2: Blocking on a queue set that contains a mutex will not cause the mutex holder to inherit the priority of the blocked task.

Note 3: A receive (in the case of a queue) or take (in the case of a semaphore) operation must not be performed on a member of a queue set unless a call to `xQueueSelectFromSet()` has first returned a handle to that set member.

Return `xQueueSelectFromSet()` will return the handle of a queue (cast to a `QueueSetMemberHandle_t` type) contained in the queue set that contains data, or the handle of a semaphore (cast to a `QueueSetMemberHandle_t` type) contained in the queue set that is available, or `NULL` if no such queue or semaphore exists before before the specified block time expires.

Parameters

- `xQueueSet`: The queue set on which the task will (potentially) block.
- `xTicksToWait`: The maximum time, in ticks, that the calling task will remain in the Blocked state (with other tasks executing) to wait for a member of the queue set to be ready for a successful queue read or semaphore take operation.

QueueSetMemberHandle_t xQueueSelectFromSetFromISR (QueueSetHandle_t xQueueSet)

A version of xQueueSelectFromSet() that can be used from an ISR.

Macros**xQueueCreate (uxQueueLength, uxItemSize)**

Creates a new queue instance. This allocates the storage required by the new queue and returns a handle for the queue.

Example usage:

```

struct AMessage
{
char ucMessageID;
char ucData[ 20 ];
};

void vATask( void *pvParameters )
{
QueueHandle_t xQueue1, xQueue2;

// Create a queue capable of containing 10 uint32_t values.
xQueue1 = xQueueCreate( 10, sizeof( uint32_t ) );
if( xQueue1 == 0 )
{
// Queue was not created and must not be used.
}

// Create a queue capable of containing 10 pointers to AMessage structures.
// These should be passed by pointer as they contain a lot of data.
xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );
if( xQueue2 == 0 )
{
// Queue was not created and must not be used.
}

// ... Rest of task code.
}

```

Return If the queue is successfully create then a handle to the newly created queue is returned. If the queue cannot be created then 0 is returned.

Parameters

- **uxQueueLength**: The maximum number of items that the queue can contain.
- **uxItemSize**: The number of bytes each item in the queue will require. Items are queued by copy, not by reference, so this is the number of bytes that will be copied for each posted item. Each item on the queue must be the same size.

xQueueCreateStatic (uxQueueLength, uxItemSize, pucQueueStorage, pxQueueBuffer)

Creates a new queue instance, and returns a handle by which the new queue can be referenced.

Internally, within the FreeRTOS implementation, queues use two blocks of memory. The first block is used to hold the queue's data structures. The second block is used to hold items placed into the queue. If a queue is created using xQueueCreate() then both blocks of memory are automatically dynamically allocated inside the xQueueCreate() function. (see <http://www.freertos.org/a00111.html>). If a queue is created using xQueueCreateStatic() then the application writer must provide the memory that will get used by the queue. xQueueCreateStatic() therefore allows a queue to be created without using any dynamic memory allocation.

<http://www.FreeRTOS.org/Embedded-RTOS-Queues.html>

Example usage:

```

struct AMessage
{

```

(continues on next page)

(continued from previous page)

```

char ucMessageID;
char ucData[ 20 ];
};

#define QUEUE_LENGTH 10
#define ITEM_SIZE sizeof( uint32_t )

// xQueueBuffer will hold the queue structure.
StaticQueue_t xQueueBuffer;

// ucQueueStorage will hold the items posted to the queue. Must be at least
// [(queue length) * ( queue item size)] bytes long.
uint8_t ucQueueStorage[ QUEUE_LENGTH * ITEM_SIZE ];

void vATask( void *pvParameters )
{
    QueueHandle_t xQueue1;

    // Create a queue capable of containing 10 uint32_t values.
    xQueue1 = xQueueCreate( QUEUE_LENGTH, // The number of items the queue can
    →hold.
                            ITEM_SIZE // The size of each item in the queue
    →hold the items in the queue.
                            &( ucQueueStorage[ 0 ] ), // The buffer that will
    →queue structure.
                            &xQueueBuffer ); // The buffer that will hold the
    →queue structure.

    // The queue is guaranteed to be created successfully as no dynamic memory
    // allocation is used. Therefore xQueue1 is now a handle to a valid queue.

    // ... Rest of task code.
}

```

Return If the queue is created then a handle to the created queue is returned. If pxQueueBuffer is NULL then NULL is returned.

Parameters

- uxQueueLength: The maximum number of items that the queue can contain.
- uxItemSize: The number of bytes each item in the queue will require. Items are queued by copy, not by reference, so this is the number of bytes that will be copied for each posted item. Each item on the queue must be the same size.
- pucQueueStorage: If uxItemSize is not zero then pucQueueStorageBuffer must point to a uint8_t array that is at least large enough to hold the maximum number of items that can be in the queue at any one time - which is (uxQueueLength * uxItemsSize) bytes. If uxItemSize is zero then pucQueueStorageBuffer can be NULL.
- pxQueueBuffer: Must point to a variable of type StaticQueue_t, which will be used to hold the queue's data structure.

xQueueSendToFront (xQueue, pvItemToQueue, xTicksToWait)

This is a macro that calls xQueueGenericSend().

Post an item to the front of a queue. The item is queued by copy, not by reference. This function must not be called from an interrupt service routine. See xQueueSendFromISR () for an alternative which may be used in an ISR.

Example usage:

```

struct AMessage
{
char ucMessageID;
char ucData[ 20 ];
} xMessage;

```

(continues on next page)

(continued from previous page)

```

uint32_t ulVar = 10UL;

void vATask( void *pvParameters )
{
QueueHandle_t xQueue1, xQueue2;
struct AMessage *pxMessage;

// Create a queue capable of containing 10 uint32_t values.
xQueue1 = xQueueCreate( 10, sizeof( uint32_t ) );

// Create a queue capable of containing 10 pointers to AMessage structures.
// These should be passed by pointer as they contain a lot of data.
xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );

// ...

if( xQueue1 != 0 )
{
    // Send an uint32_t. Wait for 10 ticks for space to become
    // available if necessary.
    if( xQueueSendToFront( xQueue1, ( void * ) &ulVar, ( TickType_t ) 10 ) != pdPASS )
    {
        // Failed to post the message, even after 10 ticks.
    }
}

if( xQueue2 != 0 )
{
    // Send a pointer to a struct AMessage object. Don't block if the
    // queue is already full.
    pxMessage = &xMessage;
    xQueueSendToFront( xQueue2, ( void * ) &pxMessage, ( TickType_t ) 0 );
}

// ... Rest of task code.
}

```

Return `pdTRUE` if the item was successfully posted, otherwise `errQUEUE_FULL`.

Parameters

- `xQueue`: The handle to the queue on which the item is to be posted.
- `pvItemToQueue`: A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from `pvItemToQueue` into the queue storage area.
- `xTicksToWait`: The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0 and the queue is full. The time is defined in tick periods so the constant `portTICK_PERIOD_MS` should be used to convert to real time if this is required.

xQueueSendToBack (`xQueue`, `pvItemToQueue`, `xTicksToWait`)

This is a macro that calls `xQueueGenericSend()`.

Post an item to the back of a queue. The item is queued by copy, not by reference. This function must not be called from an interrupt service routine. See `xQueueSendFromISR()` for an alternative which may be used in an ISR.

Example usage:

```

struct AMessage
{

```

(continues on next page)

```

char ucMessageID;
char ucData[ 20 ];
} xMessage;

uint32_t ulVar = 10UL;

void vATask( void *pvParameters )
{
QueueHandle_t xQueue1, xQueue2;
struct AMessage *pxMessage;

// Create a queue capable of containing 10 uint32_t values.
xQueue1 = xQueueCreate( 10, sizeof( uint32_t ) );

// Create a queue capable of containing 10 pointers to AMessage structures.
// These should be passed by pointer as they contain a lot of data.
xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );

// ...

if( xQueue1 != 0 )
{
    // Send an uint32_t. Wait for 10 ticks for space to become
    // available if necessary.
    if( xQueueSendToBack( xQueue1, ( void * ) &ulVar, ( TickType_t ) 10 ) != pdPASS )
    {
        // Failed to post the message, even after 10 ticks.
    }
}

if( xQueue2 != 0 )
{
    // Send a pointer to a struct AMessage object. Don't block if the
    // queue is already full.
    pxMessage = & xMessage;
    xQueueSendToBack( xQueue2, ( void * ) &pxMessage, ( TickType_t ) 0 );
}

// ... Rest of task code.
}

```

Return `pdTRUE` if the item was successfully posted, otherwise `errQUEUE_FULL`.

Parameters

- `xQueue`: The handle to the queue on which the item is to be posted.
- `pvItemToQueue`: A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from `pvItemToQueue` into the queue storage area.
- `xTicksToWait`: The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0 and the queue is full. The time is defined in tick periods so the constant `portTICK_PERIOD_MS` should be used to convert to real time if this is required.

xQueueSend (xQueue, pvItemToQueue, xTicksToWait)

This is a macro that calls `xQueueGenericSend()`. It is included for backward compatibility with versions of FreeRTOS.org that did not include the `xQueueSendToFront()` and `xQueueSendToBack()` macros. It is equivalent to `xQueueSendToBack()`.

Post an item on a queue. The item is queued by copy, not by reference. This function must not be called from an interrupt service routine. See `xQueueSendFromISR()` for an alternative which may be used in an ISR.

Example usage:

```

struct AMessage
{
char ucMessageID;
char ucData[ 20 ];
} xMessage;

uint32_t ulVar = 10UL;

void vATask( void *pvParameters )
{
QueueHandle_t xQueue1, xQueue2;
struct AMessage *pxMessage;

// Create a queue capable of containing 10 uint32_t values.
xQueue1 = xQueueCreate( 10, sizeof( uint32_t ) );

// Create a queue capable of containing 10 pointers to AMessage structures.
// These should be passed by pointer as they contain a lot of data.
xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );

// ...

if( xQueue1 != 0 )
{
// Send an uint32_t. Wait for 10 ticks for space to become
// available if necessary.
if( xQueueSend( xQueue1, ( void * ) &ulVar, ( TickType_t ) 10 ) != pdPASS )
{
// Failed to post the message, even after 10 ticks.
}
}

if( xQueue2 != 0 )
{
// Send a pointer to a struct AMessage object. Don't block if the
// queue is already full.
pxMessage = & xMessage;
xQueueSend( xQueue2, ( void * ) &pxMessage, ( TickType_t ) 0 );
}

// ... Rest of task code.
}

```

Return pdTRUE if the item was successfully posted, otherwise errQUEUE_FULL.

Parameters

- **xQueue**: The handle to the queue on which the item is to be posted.
- **pvItemToQueue**: A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
- **xTicksToWait**: The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0 and the queue is full. The time is defined in tick periods so the constant portTICK_PERIOD_MS should be used to convert to real time if this is required.

xQueueOverwrite (xQueue, pvItemToQueue)

Only for use with queues that have a length of one - so the queue is either empty or full.

Post an item on a queue. If the queue is already full then overwrite the value held in the queue. The item is queued by copy, not by reference.

This function must not be called from an interrupt service routine. See xQueueOverwriteFromISR () for an alternative which may be used in an ISR.

Example usage:

```

void vFunction( void *pvParameters )
{
    QueueHandle_t xQueue;
    uint32_t ulVarToSend, ulValReceived;

    // Create a queue to hold one uint32_t value. It is strongly
    // recommended *not* to use xQueueOverwrite() on queues that can
    // contain more than one value, and doing so will trigger an assertion
    // if configASSERT() is defined.
    xQueue = xQueueCreate( 1, sizeof( uint32_t ) );

    // Write the value 10 to the queue using xQueueOverwrite().
    ulVarToSend = 10;
    xQueueOverwrite( xQueue, &ulVarToSend );

    // Peeking the queue should now return 10, but leave the value 10 in
    // the queue. A block time of zero is used as it is known that the
    // queue holds a value.
    ulValReceived = 0;
    xQueuePeek( xQueue, &ulValReceived, 0 );

    if( ulValReceived != 10 )
    {
        // Error unless the item was removed by a different task.
    }

    // The queue is still full. Use xQueueOverwrite() to overwrite the
    // value held in the queue with 100.
    ulVarToSend = 100;
    xQueueOverwrite( xQueue, &ulVarToSend );

    // This time read from the queue, leaving the queue empty once more.
    // A block time of 0 is used again.
    xQueueReceive( xQueue, &ulValReceived, 0 );

    // The value read should be the last value written, even though the
    // queue was already full when the value was written.
    if( ulValReceived != 100 )
    {
        // Error!
    }

    // ...
}

```

Return `xQueueOverwrite()` is a macro that calls `xQueueGenericSend()`, and therefore has the same return values as `xQueueSendToFront()`. However, `pdPASS` is the only value that can be returned because `xQueueOverwrite()` will write to the queue even when the queue is already full.

Parameters

- `xQueue`: The handle of the queue to which the data is being sent.
- `pvItemToQueue`: A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from `pvItemToQueue` into the queue storage area.

`xQueuePeek` (`xQueue`, `pvBuffer`, `xTicksToWait`)

This is a macro that calls the `xQueueGenericReceive()` function.

Receive an item from a queue without removing the item from the queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created.

Successfully received items remain on the queue so will be returned again by the next call, or a call to `xQueueReceive()`.

This macro must not be used in an interrupt service routine. See `xQueuePeekFromISR()` for an alternative that can be called from an interrupt service routine.

Example usage:

```

struct AMessage
{
char ucMessageID;
char ucData[ 20 ];
} xMessage;

QueueHandle_t xQueue;

// Task to create a queue and post a value.
void vATask( void *pvParameters )
{
struct AMessage *pxMessage;

// Create a queue capable of containing 10 pointers to AMessage structures.
// These should be passed by pointer as they contain a lot of data.
xQueue = xQueueCreate( 10, sizeof( struct AMessage * ) );
if( xQueue == 0 )
{
// Failed to create the queue.
}

// ...

// Send a pointer to a struct AMessage object. Don't block if the
// queue is already full.
pxMessage = & xMessage;
xQueueSend( xQueue, ( void * ) &pxMessage, ( TickType_t ) 0 );

// ... Rest of task code.
}

// Task to peek the data from the queue.
void vADifferentTask( void *pvParameters )
{
struct AMessage *pRxedMessage;

if( xQueue != 0 )
{
// Peek a message on the created queue. Block for 10 ticks if a
// message is not immediately available.
if( xQueuePeek( xQueue, &( pRxedMessage ), ( TickType_t ) 10 ) )
{
// pRxedMessage now points to the struct AMessage variable posted
// by vATask, but the item still remains on the queue.
}
}

// ... Rest of task code.
}

```

Return `pdTRUE` if an item was successfully received from the queue, otherwise `pdFALSE`.

Parameters

- `xQueue`: The handle to the queue from which the item is to be received.
- `pvBuffer`: Pointer to the buffer into which the received item will be copied.
- `xTicksToWait`: The maximum amount of time the task should block waiting for an item to

receive should the queue be empty at the time of the call. The time is defined in tick periods so the constant `portTICK_PERIOD_MS` should be used to convert to real time if this is required. `xQueuePeek()` will return immediately if `xTicksToWait` is 0 and the queue is empty.

xQueueReceive (xQueue, pvBuffer, xTicksToWait)

queue. h

This is a macro that calls the `xQueueGenericReceive()` function.

Receive an item from a queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created.

Successfully received items are removed from the queue.

This function must not be used in an interrupt service routine. See `xQueueReceiveFromISR` for an alternative that can.

Example usage:

```

struct AMessage
{
    char ucMessageID;
    char ucData[ 20 ];
} xMessage;

QueueHandle_t xQueue;

// Task to create a queue and post a value.
void vATask( void *pvParameters )
{
    struct AMessage *pxMessage;

    // Create a queue capable of containing 10 pointers to AMessage structures.
    // These should be passed by pointer as they contain a lot of data.
    xQueue = xQueueCreate( 10, sizeof( struct AMessage * ) );
    if( xQueue == 0 )
    {
        // Failed to create the queue.
    }

    // ...

    // Send a pointer to a struct AMessage object. Don't block if the
    // queue is already full.
    pxMessage = & xMessage;
    xQueueSend( xQueue, ( void * ) &pxMessage, ( TickType_t ) 0 );

    // ... Rest of task code.
}

// Task to receive from the queue.
void vADifferentTask( void *pvParameters )
{
    struct AMessage *pRxedMessage;

    if( xQueue != 0 )
    {
        // Receive a message on the created queue. Block for 10 ticks if a
        // message is not immediately available.
        if( xQueueReceive( xQueue, &( pRxedMessage ), ( TickType_t ) 10 ) )
        {
            // pRxedMessage now points to the struct AMessage variable posted
            // by vATask.
        }
    }
}

```

(continues on next page)

(continued from previous page)

```

}

// ... Rest of task code.
}

```

Return pdTRUE if an item was successfully received from the queue, otherwise pdFALSE.

Parameters

- **xQueue**: The handle to the queue from which the item is to be received.
- **pvBuffer**: Pointer to the buffer into which the received item will be copied.
- **xTicksToWait**: The maximum amount of time the task should block waiting for an item to receive should the queue be empty at the time of the call. xQueueReceive() will return immediately if xTicksToWait is zero and the queue is empty. The time is defined in tick periods so the constant portTICK_PERIOD_MS should be used to convert to real time if this is required.

xQueueSendToFrontFromISR (xQueue, pvItemToQueue, pxHigherPriorityTaskWoken)

This is a macro that calls xQueueGenericSendFromISR().

Post an item to the front of a queue. It is safe to use this macro from within an interrupt service routine.

Items are queued by copy not reference so it is preferable to only queue small items, especially when called from an ISR. In most cases it would be preferable to store a pointer to the item being queued.

Example usage for buffered IO (where the ISR can obtain more than one value per call):

```

void vBufferISR( void )
{
char cIn;
BaseType_t xHigherPriorityTaskWoken;

// We have not woken a task at the start of the ISR.
xHigherPriorityTaskWoken = pdFALSE;

// Loop until the buffer is empty.
do
{
// Obtain a byte from the buffer.
cIn = portINPUT_BYTE( RX_REGISTER_ADDRESS );

// Post the byte.
xQueueSendToFrontFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWoken );

} while( portINPUT_BYTE( BUFFER_COUNT ) );

// Now the buffer is empty we can switch context if necessary.
if( xHigherPriorityTaskWoken )
{
portYIELD_FROM_ISR ();
}
}

```

Return pdTRUE if the data was successfully sent to the queue, otherwise errQUEUE_FULL.

Parameters

- **xQueue**: The handle to the queue on which the item is to be posted.
- **pvItemToQueue**: A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
- [out] **pxHigherPriorityTaskWoken**: xQueueSendToFrontFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE if sending to the queue caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If xQueueSendToFromFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited.

xQueueSendToBackFromISR (xQueue, pvItemToQueue, pxHigherPriorityTaskWoken)

This is a macro that calls `xQueueGenericSendFromISR()`.

Post an item to the back of a queue. It is safe to use this macro from within an interrupt service routine.

Items are queued by copy not reference so it is preferable to only queue small items, especially when called from an ISR. In most cases it would be preferable to store a pointer to the item being queued.

Example usage for buffered IO (where the ISR can obtain more than one value per call):

```
void vBufferISR( void )
{
char cIn;
 BaseType_t xHigherPriorityTaskWoken;

// We have not woken a task at the start of the ISR.
xHigherPriorityTaskWoken = pdFALSE;

// Loop until the buffer is empty.
do
{
// Obtain a byte from the buffer.
cIn = portINPUT_BYTE( RX_REGISTER_ADDRESS );

// Post the byte.
xQueueSendToBackFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWoken );

} while( portINPUT_BYTE( BUFFER_COUNT ) );

// Now the buffer is empty we can switch context if necessary.
if( xHigherPriorityTaskWoken )
{
portYIELD_FROM_ISR ();
}
}
```

Return `pdTRUE` if the data was successfully sent to the queue, otherwise `errQUEUE_FULL`.

Parameters

- `xQueue`: The handle to the queue on which the item is to be posted.
- `pvItemToQueue`: A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from `pvItemToQueue` into the queue storage area.
- [out] `pxHigherPriorityTaskWoken`: `xQueueSendToBackFromISR()` will set `*pxHigherPriorityTaskWoken` to `pdTRUE` if sending to the queue caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If `xQueueSendToBackFromISR()` sets this value to `pdTRUE` then a context switch should be requested before the interrupt is exited.

xQueueOverwriteFromISR(`xQueue`, `pvItemToQueue`, `pxHigherPriorityTaskWoken`)

A version of `xQueueOverwrite()` that can be used in an interrupt service routine (ISR).

Only for use with queues that can hold a single item - so the queue is either empty or full.

Post an item on a queue. If the queue is already full then overwrite the value held in the queue. The item is queued by copy, not by reference.

Example usage:

```
QueueHandle_t xQueue;

void vFunction( void *pvParameters )
{
// Create a queue to hold one uint32_t value. It is strongly
// recommended *not* to use xQueueOverwriteFromISR() on queues that can
// contain more than one value, and doing so will trigger an assertion
// if configASSERT() is defined.
```

(continues on next page)

(continued from previous page)

```

    xQueue = xQueueCreate( 1, sizeof( uint32_t ) );
}

void vAnInterruptHandler( void )
{
// xHigherPriorityTaskWoken must be set to pdFALSE before it is used.
BaseType_t xHigherPriorityTaskWoken = pdFALSE;
uint32_t ulVarToSend, ulValReceived;

// Write the value 10 to the queue using xQueueOverwriteFromISR().
ulVarToSend = 10;
xQueueOverwriteFromISR( xQueue, &ulVarToSend, &xHigherPriorityTaskWoken );

// The queue is full, but calling xQueueOverwriteFromISR() again will still
// pass because the value held in the queue will be overwritten with the
// new value.
ulVarToSend = 100;
xQueueOverwriteFromISR( xQueue, &ulVarToSend, &xHigherPriorityTaskWoken );

// Reading from the queue will now return 100.

// ...

if( xHigherPriorityTaskWoken == pdTRUE )
{
// Writing to the queue caused a task to unblock and the unblocked task
// has a priority higher than or equal to the priority of the currently
// executing task (the task this interrupt interrupted). Perform a
↪context
// switch so this interrupt returns directly to the unblocked task.
portYIELD_FROM_ISR(); // or portEND_SWITCHING_ISR() depending on the
↪port.
}
}

```

Return `xQueueOverwriteFromISR()` is a macro that calls `xQueueGenericSendFromISR()`, and therefore has the same return values as `xQueueSendToFrontFromISR()`. However, `pdPASS` is the only value that can be returned because `xQueueOverwriteFromISR()` will write to the queue even when the queue is already full.

Parameters

- `xQueue`: The handle to the queue on which the item is to be posted.
- `pvItemToQueue`: A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from `pvItemToQueue` into the queue storage area.
- [out] `pxHigherPriorityTaskWoken`: `xQueueOverwriteFromISR()` will set `*pxHigherPriorityTaskWoken` to `pdTRUE` if sending to the queue caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If `xQueueOverwriteFromISR()` sets this value to `pdTRUE` then a context switch should be requested before the interrupt is exited.

xQueueSendFromISR (`xQueue`, `pvItemToQueue`, `pxHigherPriorityTaskWoken`)

This is a macro that calls `xQueueGenericSendFromISR()`. It is included for backward compatibility with versions of FreeRTOS.org that did not include the `xQueueSendToBackFromISR()` and `xQueueSendToFrontFromISR()` macros.

Post an item to the back of a queue. It is safe to use this function from within an interrupt service routine.

Items are queued by copy not reference so it is preferable to only queue small items, especially when called from an ISR. In most cases it would be preferable to store a pointer to the item being queued.

Example usage for buffered IO (where the ISR can obtain more than one value per call):

```

void vBufferISR( void )
{
char cIn;
BaseType_t xHigherPriorityTaskWoken;

// We have not woken a task at the start of the ISR.
xHigherPriorityTaskWoken = pdFALSE;

// Loop until the buffer is empty.
do
{
// Obtain a byte from the buffer.
cIn = portINPUT_BYTE( RX_REGISTER_ADDRESS );

// Post the byte.
xQueueSendFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWoken );

} while( portINPUT_BYTE( BUFFER_COUNT ) );

// Now the buffer is empty we can switch context if necessary.
if( xHigherPriorityTaskWoken )
{
// Actual macro used here is port specific.
portYIELD_FROM_ISR ();
}
}

```

Return pdTRUE if the data was successfully sent to the queue, otherwise errQUEUE_FULL.

Parameters

- xQueue: The handle to the queue on which the item is to be posted.
- pvItemToQueue: A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
- [out] pxHigherPriorityTaskWoken: xQueueSendFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE if sending to the queue caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If xQueueSendFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited.

xQueueReset (xQueue)

Reset a queue back to its original empty state. pdPASS is returned if the queue is successfully reset. pdFAIL is returned if the queue could not be reset because there are tasks blocked on the queue waiting to either receive from the queue or send to the queue.

Return always returns pdPASS

Parameters

- xQueue: The queue to reset

Type Definitions

typedef void *QueueHandle_t

Type by which queues are referenced. For example, a call to xQueueCreate() returns an QueueHandle_t variable that can then be used as a parameter to xQueueSend(), xQueueReceive(), etc.

typedef void *QueueSetHandle_t

Type by which queue sets are referenced. For example, a call to xQueueCreateSet() returns an xQueueSet variable that can then be used as a parameter to xQueueSelectFromSet(), xQueueAddToSet(), etc.

typedef void *QueueSetMemberHandle_t

Queue sets can contain both queues and semaphores, so the QueueSetMemberHandle_t is defined as a type to be used where a parameter or return value can be either an QueueHandle_t or an SemaphoreHandle_t.

Semaphore API

Header File

- [freertos/include/freertos/semphr.h](#)

Macros

semBINARY_SEMAPHORE_QUEUE_LENGTH

semSEMAPHORE_QUEUE_ITEM_LENGTH

semGIVE_BLOCK_TIME

xSemaphoreCreateBinary()

Creates a new binary semaphore instance, and returns a handle by which the new semaphore can be referenced.

In many usage scenarios it is faster and more memory efficient to use a direct to task notification in place of a binary semaphore! <http://www.freertos.org/RTOS-task-notifications.html>

Internally, within the FreeRTOS implementation, binary semaphores use a block of memory, in which the semaphore structure is stored. If a binary semaphore is created using `xSemaphoreCreateBinary()` then the required memory is automatically dynamically allocated inside the `xSemaphoreCreateBinary()` function. (see <http://www.freertos.org/a00111.html>). If a binary semaphore is created using `xSemaphoreCreateBinaryStatic()` then the application writer must provide the memory. `xSemaphoreCreateBinaryStatic()` therefore allows a binary semaphore to be created without using any dynamic memory allocation.

The old `vSemaphoreCreateBinary()` macro is now deprecated in favour of this `xSemaphoreCreateBinary()` function. Note that binary semaphores created using the `vSemaphoreCreateBinary()` macro are created in a state such that the first call to ‘take’ the semaphore would pass, whereas binary semaphores created using `xSemaphoreCreateBinary()` are created in a state such that the the semaphore must first be ‘given’ before it can be ‘taken’.

Function that creates a semaphore by using the existing queue mechanism. The queue length is 1 as this is a binary semaphore. The data size is 0 as nothing is actually stored - all that is important is whether the queue is empty or full (the binary semaphore is available or not).

This type of semaphore can be used for pure synchronisation between tasks or between an interrupt and a task. The semaphore need not be given back once obtained, so one task/interrupt can continuously ‘give’ the semaphore while another continuously ‘takes’ the semaphore. For this reason this type of semaphore does not use a priority inheritance mechanism. For an alternative that does use priority inheritance see `xSemaphoreCreateMutex()`.

Example usage:

```
SemaphoreHandle_t xSemaphore = NULL;

void vATask( void * pvParameters )
{
    // Semaphore cannot be used before a call to vSemaphoreCreateBinary ().
    // This is a macro so pass the variable in directly.
    xSemaphore = xSemaphoreCreateBinary();

    if( xSemaphore != NULL )
    {
        // The semaphore was created successfully.
        // The semaphore can now be used.
    }
}
```

Return Handle to the created semaphore.

xSemaphoreCreateBinaryStatic (pxStaticSemaphore)

Creates a new binary semaphore instance, and returns a handle by which the new semaphore can be referenced.

NOTE: In many usage scenarios it is faster and more memory efficient to use a direct to task notification in place of a binary semaphore! <http://www.freertos.org/RTOS-task-notifications.html>

Internally, within the FreeRTOS implementation, binary semaphores use a block of memory, in which the semaphore structure is stored. If a binary semaphore is created using `xSemaphoreCreateBinary()` then the required memory is automatically dynamically allocated inside the `xSemaphoreCreateBinary()` function. (see <http://www.freertos.org/a00111.html>). If a binary semaphore is created using `xSemaphoreCreateBinaryStatic()` then the application writer must provide the memory. `xSemaphoreCreateBinaryStatic()` therefore allows a binary semaphore to be created without using any dynamic memory allocation.

This type of semaphore can be used for pure synchronisation between tasks or between an interrupt and a task. The semaphore need not be given back once obtained, so one task/interrupt can continuously ‘give’ the semaphore while another continuously ‘takes’ the semaphore. For this reason this type of semaphore does not use a priority inheritance mechanism. For an alternative that does use priority inheritance see `xSemaphoreCreateMutex()`.

Example usage:

```
SemaphoreHandle_t xSemaphore = NULL;
StaticSemaphore_t xSemaphoreBuffer;

void vATask( void * pvParameters )
{
    // Semaphore cannot be used before a call to xSemaphoreCreateBinary().
    // The semaphore's data structures will be placed in the xSemaphoreBuffer
    // variable, the address of which is passed into the function. The
    // function's parameter is not NULL, so the function will not attempt any
    // dynamic memory allocation, and therefore the function will not return
    // return NULL.
    xSemaphore = xSemaphoreCreateBinary( &xSemaphoreBuffer );

    // Rest of task code goes here.
}
```

Return If the semaphore is created then a handle to the created semaphore is returned. If `pxSemaphoreBuffer` is NULL then NULL is returned.

Parameters

- `pxStaticSemaphore`: Must point to a variable of type `StaticSemaphore_t`, which will then be used to hold the semaphore's data structure, removing the need for the memory to be allocated dynamically.

xSemaphoreTake (xSemaphore, xBlockTime)

Macro to obtain a semaphore. The semaphore must have previously been created with a call to `vSemaphoreCreateBinary()`, `xSemaphoreCreateMutex()` or `xSemaphoreCreateCounting()`.

Example usage:

```
SemaphoreHandle_t xSemaphore = NULL;

// A task that creates a semaphore.
void vATask( void * pvParameters )
{
    // Create the semaphore to guard a shared resource.
    vSemaphoreCreateBinary( xSemaphore );
}

// A task that uses the semaphore.
void vAnotherTask( void * pvParameters )
{
    // ... Do other things.

    if( xSemaphore != NULL )
    {
```

(continues on next page)

(continued from previous page)

```

// See if we can obtain the semaphore. If the semaphore is not
↪available
// wait 10 ticks to see if it becomes free.
if( xSemaphoreTake( xSemaphore, ( TickType_t ) 10 ) == pdTRUE )
{
    // We were able to obtain the semaphore and can now access the
    // shared resource.

    // ...

    // We have finished accessing the shared resource. Release the
    // semaphore.
    xSemaphoreGive( xSemaphore );
}
else
{
    // We could not obtain the semaphore and can therefore not access
    // the shared resource safely.
}
}
}

```

Return pdTRUE if the semaphore was obtained. pdFALSE if xBlockTime expired without the semaphore becoming available.

Parameters

- xSemaphore: A handle to the semaphore being taken - obtained when the semaphore was created.
- xBlockTime: The time in ticks to wait for the semaphore to become available. The macro portTICK_PERIOD_MS can be used to convert this to a real time. A block time of zero can be used to poll the semaphore. A block time of portMAX_DELAY can be used to block indefinitely (provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h).

xSemaphoreTakeRecursive (xMutex, xBlockTime)

Macro to recursively obtain, or ‘take’, a mutex type semaphore. The mutex must have previously been created using a call to xSemaphoreCreateRecursiveMutex();

configUSE_RECURSIVE_MUTEXES must be set to 1 in FreeRTOSConfig.h for this macro to be available.

This macro must not be used on mutexes created using xSemaphoreCreateMutex().

A mutex used recursively can be ‘taken’ repeatedly by the owner. The mutex doesn’t become available again until the owner has called xSemaphoreGiveRecursive() for each successful ‘take’ request. For example, if a task successfully ‘takes’ the same mutex 5 times then the mutex will not be available to any other task until it has also ‘given’ the mutex back exactly five times.

Example usage:

```

SemaphoreHandle_t xMutex = NULL;

// A task that creates a mutex.
void vATask( void * pvParameters )
{
    // Create the mutex to guard a shared resource.
    xMutex = xSemaphoreCreateRecursiveMutex();
}

// A task that uses the mutex.
void vAnotherTask( void * pvParameters )
{
    // ... Do other things.

    if( xMutex != NULL )
    {

```

(continues on next page)

(continued from previous page)

```

// See if we can obtain the mutex. If the mutex is not available
// wait 10 ticks to see if it becomes free.
if( xSemaphoreTakeRecursive( xSemaphore, ( TickType_t ) 10 ) == pdTRUE )
{
    // We were able to obtain the mutex and can now access the
    // shared resource.

    // ...
    // For some reason due to the nature of the code further calls to
    // xSemaphoreTakeRecursive() are made on the same mutex. In real
    // code these would not be just sequential calls as this would make
    // no sense. Instead the calls are likely to be buried inside
    // a more complex call structure.
    xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 );
    xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 );

    // The mutex has now been 'taken' three times, so will not be
    // available to another task until it has also been given back
    // three times. Again it is unlikely that real code would have
    // these calls sequentially, but instead buried in a more complex
    // call structure. This is just for illustrative purposes.
    xSemaphoreGiveRecursive( xMutex );
    xSemaphoreGiveRecursive( xMutex );
    xSemaphoreGiveRecursive( xMutex );

    // Now the mutex can be taken by other tasks.
}
else
{
    // We could not obtain the mutex and can therefore not access
    // the shared resource safely.
}
}
}

```

Return pdTRUE if the semaphore was obtained. pdFALSE if xBlockTime expired without the semaphore becoming available.

Parameters

- **xMutex**: A handle to the mutex being obtained. This is the handle returned by xSemaphoreCreateRecursiveMutex();
- **xBlockTime**: The time in ticks to wait for the semaphore to become available. The macro portTICK_PERIOD_MS can be used to convert this to a real time. A block time of zero can be used to poll the semaphore. If the task already owns the semaphore then xSemaphoreTakeRecursive() will return immediately no matter what the value of xBlockTime.

xSemaphoreGive (xSemaphore)

Macro to release a semaphore. The semaphore must have previously been created with a call to vSemaphoreCreateBinary(), xSemaphoreCreateMutex() or xSemaphoreCreateCounting(). and obtained using sSemaphoreTake().

This macro must not be used from an ISR. See xSemaphoreGiveFromISR () for an alternative which can be used from an ISR.

This macro must also not be used on semaphores created using xSemaphoreCreateRecursiveMutex().

Example usage:

```

SemaphoreHandle_t xSemaphore = NULL;

void vATask( void * pvParameters )
{

```

(continues on next page)

(continued from previous page)

```

// Create the semaphore to guard a shared resource.
vSemaphoreCreateBinary( xSemaphore );

if( xSemaphore != NULL )
{
    if( xSemaphoreGive( xSemaphore ) != pdTRUE )
    {
        // We would expect this call to fail because we cannot give
        // a semaphore without first "taking" it!
    }

    // Obtain the semaphore - don't block if the semaphore is not
    // immediately available.
    if( xSemaphoreTake( xSemaphore, ( TickType_t ) 0 ) )
    {
        // We now have the semaphore and can access the shared resource.

        // ...

        // We have finished accessing the shared resource so can free the
        // semaphore.
        if( xSemaphoreGive( xSemaphore ) != pdTRUE )
        {
            // We would not expect this call to fail because we must have
            // obtained the semaphore to get here.
        }
    }
}
}

```

Return pdTRUE if the semaphore was released. pdFALSE if an error occurred. Semaphores are implemented using queues. An error can occur if there is no space on the queue to post a message - indicating that the semaphore was not first obtained correctly.

Parameters

- **xSemaphore**: A handle to the semaphore being released. This is the handle returned when the semaphore was created.

xSemaphoreGiveRecursive(xMutex)

Macro to recursively release, or ‘give’, a mutex type semaphore. The mutex must have previously been created using a call to xSemaphoreCreateRecursiveMutex();

configUSE_RECURSIVE_MUTEXES must be set to 1 in FreeRTOSConfig.h for this macro to be available.

This macro must not be used on mutexes created using xSemaphoreCreateMutex().

A mutex used recursively can be ‘taken’ repeatedly by the owner. The mutex doesn’t become available again until the owner has called xSemaphoreGiveRecursive() for each successful ‘take’ request. For example, if a task successfully ‘takes’ the same mutex 5 times then the mutex will not be available to any other task until it has also ‘given’ the mutex back exactly five times.

Example usage:

```

SemaphoreHandle_t xMutex = NULL;

// A task that creates a mutex.
void vATask( void * pvParameters )
{
    // Create the mutex to guard a shared resource.
    xMutex = xSemaphoreCreateRecursiveMutex();
}

// A task that uses the mutex.

```

(continues on next page)

(continued from previous page)

```

void vAnotherTask( void * pvParameters )
{
    // ... Do other things.

    if( xMutex != NULL )
    {
        // See if we can obtain the mutex.  If the mutex is not available
        // wait 10 ticks to see if it becomes free.
        if( xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 ) == pdTRUE )
        {
            // We were able to obtain the mutex and can now access the
            // shared resource.

            // ...
            // For some reason due to the nature of the code further calls to
            // xSemaphoreTakeRecursive() are made on the same mutex.  In real
            // code these would not be just sequential calls as this would make
            // no sense.  Instead the calls are likely to be buried inside
            // a more complex call structure.
            xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 );
            xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 );

            // The mutex has now been 'taken' three times, so will not be
            // available to another task until it has also been given back
            // three times.  Again it is unlikely that real code would have
            // these calls sequentially, it would be more likely that the calls
            // to xSemaphoreGiveRecursive() would be called as a call stack
            // unwound.  This is just for demonstrative purposes.
            xSemaphoreGiveRecursive( xMutex );
            xSemaphoreGiveRecursive( xMutex );
            xSemaphoreGiveRecursive( xMutex );

            // Now the mutex can be taken by other tasks.
        }
        else
        {
            // We could not obtain the mutex and can therefore not access
            // the shared resource safely.
        }
    }
}

```

Return pdTRUE if the semaphore was given.

Parameters

- **xMutex**: A handle to the mutex being released, or 'given'. This is the handle returned by xSemaphoreCreateMutex();

xSemaphoreGiveFromISR (xSemaphore, pxHigherPriorityTaskWoken)

Macro to release a semaphore. The semaphore must have previously been created with a call to vSemaphoreCreateBinary() or xSemaphoreCreateCounting().

Mutex type semaphores (those created using a call to xSemaphoreCreateMutex()) must not be used with this macro.

This macro can be used from an ISR.

Example usage:

```

#define LONG_TIME 0xffff
#define TICKS_TO_WAIT 10
SemaphoreHandle_t xSemaphore = NULL;

```

(continues on next page)

(continued from previous page)

```
// Repetitive task.
void vATask( void * pvParameters )
{
    for( ;; )
    {
        // We want this task to run every 10 ticks of a timer.  The semaphore
        // was created before this task was started.

        // Block waiting for the semaphore to become available.
        if( xSemaphoreTake( xSemaphore, LONG_TIME ) == pdTRUE )
        {
            // It is time to execute.

            // ...

            // We have finished our task.  Return to the top of the loop where
            // we will block on the semaphore until it is time to execute
            // again.  Note when using the semaphore for synchronisation with an
            // ISR in this manner there is no need to 'give' the semaphore back.
        }
    }
}

// Timer ISR
void vTimerISR( void * pvParameters )
{
    static uint8_t ucLocalTickCount = 0;
    static BaseType_t xHigherPriorityTaskWoken;

    // A timer tick has occurred.

    // ... Do other time functions.

    // Is it time for vATask () to run?
    xHigherPriorityTaskWoken = pdFALSE;
    ucLocalTickCount++;
    if( ucLocalTickCount >= TICKS_TO_WAIT )
    {
        // Unblock the task by releasing the semaphore.
        xSemaphoreGiveFromISR( xSemaphore, &xHigherPriorityTaskWoken );

        // Reset the count so we release the semaphore again in 10 ticks time.
        ucLocalTickCount = 0;
    }

    if( xHigherPriorityTaskWoken != pdFALSE )
    {
        // We can force a context switch here.  Context switching from an
        // ISR uses port specific syntax.  Check the demo task for your port
        // to find the syntax required.
    }
}
```

Return pdTRUE if the semaphore was successfully given, otherwise errQUEUE_FULL.

Parameters

- xSemaphore: A handle to the semaphore being released. This is the handle returned when the semaphore was created.
- [out] pxHigherPriorityTaskWoken: xSemaphoreGiveFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE if giving the semaphore caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If xSemaphoreGiveFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited.

xSemaphoreTakeFromISR (xSemaphore, pxHigherPriorityTaskWoken)

Macro to take a semaphore from an ISR. The semaphore must have previously been created with a call to vSemaphoreCreateBinary() or xSemaphoreCreateCounting().

Mutex type semaphores (those created using a call to xSemaphoreCreateMutex()) must not be used with this macro.

This macro can be used from an ISR, however taking a semaphore from an ISR is not a common operation. It is likely to only be useful when taking a counting semaphore when an interrupt is obtaining an object from a resource pool (when the semaphore count indicates the number of resources available).

Return pdTRUE if the semaphore was successfully taken, otherwise pdFALSE

Parameters

- xSemaphore: A handle to the semaphore being taken. This is the handle returned when the semaphore was created.
- [out] pxHigherPriorityTaskWoken: xSemaphoreTakeFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE if taking the semaphore caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If xSemaphoreTakeFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited.

xSemaphoreCreateMutex ()

Macro that implements a mutex semaphore by using the existing queue mechanism.

Internally, within the FreeRTOS implementation, mutex semaphores use a block of memory, in which the mutex structure is stored. If a mutex is created using xSemaphoreCreateMutex() then the required memory is automatically dynamically allocated inside the xSemaphoreCreateMutex() function. (see <http://www.freertos.org/a00111.html>). If a mutex is created using xSemaphoreCreateMutexStatic() then the application writer must provide the memory. xSemaphoreCreateMutexStatic() therefore allows a mutex to be created without using any dynamic memory allocation.

Mutexes created using this function can be accessed using the xSemaphoreTake() and xSemaphoreGive() macros. The xSemaphoreTakeRecursive() and xSemaphoreGiveRecursive() macros must not be used.

This type of semaphore uses a priority inheritance mechanism so a task ‘taking’ a semaphore MUST ALWAYS ‘give’ the semaphore back once the semaphore it is no longer required.

Mutex type semaphores cannot be used from within interrupt service routines.

See vSemaphoreCreateBinary() for an alternative implementation that can be used for pure synchronisation (where one task or interrupt always ‘gives’ the semaphore and another always ‘takes’ the semaphore) and from within interrupt service routines.

Example usage:

```
SemaphoreHandle_t xSemaphore;

void vATask( void * pvParameters )
{
    // Semaphore cannot be used before a call to xSemaphoreCreateMutex().
    // This is a macro so pass the variable in directly.
    xSemaphore = xSemaphoreCreateMutex();

    if( xSemaphore != NULL )
    {
        // The semaphore was created successfully.
        // The semaphore can now be used.
    }
}
```

Return If the mutex was successfully created then a handle to the created semaphore is returned. If there was not enough heap to allocate the mutex data structures then NULL is returned.

xSemaphoreCreateMutexStatic (pxMutexBuffer)

Creates a new mutex type semaphore instance, and returns a handle by which the new mutex can be referenced.

Internally, within the FreeRTOS implementation, mutex semaphores use a block of memory, in which the mutex structure is stored. If a mutex is created using `xSemaphoreCreateMutex()` then the required memory is automatically dynamically allocated inside the `xSemaphoreCreateMutex()` function. (see <http://www.freertos.org/a00111.html>). If a mutex is created using `xSemaphoreCreateMutexStatic()` then the application writer must provide the memory. `xSemaphoreCreateMutexStatic()` therefore allows a mutex to be created without using any dynamic memory allocation.

Mutexes created using this function can be accessed using the `xSemaphoreTake()` and `xSemaphoreGive()` macros. The `xSemaphoreTakeRecursive()` and `xSemaphoreGiveRecursive()` macros must not be used.

This type of semaphore uses a priority inheritance mechanism so a task ‘taking’ a semaphore MUST ALWAYS ‘give’ the semaphore back once the semaphore it is no longer required.

Mutex type semaphores cannot be used from within interrupt service routines.

See `xSemaphoreCreateBinary()` for an alternative implementation that can be used for pure synchronisation (where one task or interrupt always ‘gives’ the semaphore and another always ‘takes’ the semaphore) and from within interrupt service routines.

Example usage:

```
SemaphoreHandle_t xSemaphore;
StaticSemaphore_t xMutexBuffer;

void vATask( void * pvParameters )
{
    // A mutex cannot be used before it has been created. xMutexBuffer is
    // into xSemaphoreCreateMutexStatic() so no dynamic memory allocation is
    // attempted.
    xSemaphore = xSemaphoreCreateMutexStatic( &xMutexBuffer );

    // As no dynamic memory allocation was performed, xSemaphore cannot be NULL,
    // so there is no need to check it.
}
```

Return If the mutex was successfully created then a handle to the created mutex is returned. If `pxMutexBuffer` was NULL then NULL is returned.

Parameters

- `pxMutexBuffer`: Must point to a variable of type `StaticSemaphore_t`, which will be used to hold the mutex’s data structure, removing the need for the memory to be allocated dynamically.

xSemaphoreCreateRecursiveMutex()

Creates a new recursive mutex type semaphore instance, and returns a handle by which the new recursive mutex can be referenced.

Internally, within the FreeRTOS implementation, recursive mutexes use a block of memory, in which the mutex structure is stored. If a recursive mutex is created using `xSemaphoreCreateRecursiveMutex()` then the required memory is automatically dynamically allocated inside the `xSemaphoreCreateRecursiveMutex()` function. (see <http://www.freertos.org/a00111.html>). If a recursive mutex is created using `xSemaphoreCreateRecursiveMutexStatic()` then the application writer must provide the memory that will get used by the mutex. `xSemaphoreCreateRecursiveMutexStatic()` therefore allows a recursive mutex to be created without using any dynamic memory allocation.

Mutexes created using this macro can be accessed using the `xSemaphoreTakeRecursive()` and `xSemaphoreGiveRecursive()` macros. The `xSemaphoreTake()` and `xSemaphoreGive()` macros must not be used.

A mutex used recursively can be ‘taken’ repeatedly by the owner. The mutex doesn’t become available again until the owner has called `xSemaphoreGiveRecursive()` for each successful ‘take’ request. For example, if a task successfully ‘takes’ the same mutex 5 times then the mutex will not be available to any other task until it has also ‘given’ the mutex back exactly five times.

This type of semaphore uses a priority inheritance mechanism so a task ‘taking’ a semaphore MUST ALWAYS ‘give’ the semaphore back once the semaphore it is no longer required.

Mutex type semaphores cannot be used from within interrupt service routines.

See `vSemaphoreCreateBinary()` for an alternative implementation that can be used for pure synchronisation (where one task or interrupt always ‘gives’ the semaphore and another always ‘takes’ the semaphore) and from within interrupt service routines.

Example usage:

```
SemaphoreHandle_t xSemaphore;

void vATask( void * pvParameters )
{
    // Semaphore cannot be used before a call to xSemaphoreCreateMutex().
    // This is a macro so pass the variable in directly.
    xSemaphore = xSemaphoreCreateRecursiveMutex();

    if( xSemaphore != NULL )
    {
        // The semaphore was created successfully.
        // The semaphore can now be used.
    }
}
```

Return xSemaphore Handle to the created mutex semaphore. Should be of type `SemaphoreHandle_t`.

xSemaphoreCreateRecursiveMutexStatic (pxStaticSemaphore)

Creates a new recursive mutex type semaphore instance, and returns a handle by which the new recursive mutex can be referenced.

Internally, within the FreeRTOS implementation, recursive mutexes use a block of memory, in which the mutex structure is stored. If a recursive mutex is created using `xSemaphoreCreateRecursiveMutex()` then the required memory is automatically dynamically allocated inside the `xSemaphoreCreateRecursiveMutex()` function. (see <http://www.freertos.org/a00111.html>). If a recursive mutex is created using `xSemaphoreCreateRecursiveMutexStatic()` then the application writer must provide the memory that will get used by the mutex. `xSemaphoreCreateRecursiveMutexStatic()` therefore allows a recursive mutex to be created without using any dynamic memory allocation.

Mutexes created using this macro can be accessed using the `xSemaphoreTakeRecursive()` and `xSemaphoreGiveRecursive()` macros. The `xSemaphoreTake()` and `xSemaphoreGive()` macros must not be used.

A mutex used recursively can be ‘taken’ repeatedly by the owner. The mutex doesn’t become available again until the owner has called `xSemaphoreGiveRecursive()` for each successful ‘take’ request. For example, if a task successfully ‘takes’ the same mutex 5 times then the mutex will not be available to any other task until it has also ‘given’ the mutex back exactly five times.

This type of semaphore uses a priority inheritance mechanism so a task ‘taking’ a semaphore **MUST ALWAYS** ‘give’ the semaphore back once the semaphore it is no longer required.

Mutex type semaphores cannot be used from within interrupt service routines.

See `xSemaphoreCreateBinary()` for an alternative implementation that can be used for pure synchronisation (where one task or interrupt always ‘gives’ the semaphore and another always ‘takes’ the semaphore) and from within interrupt service routines.

Example usage:

```
SemaphoreHandle_t xSemaphore;
StaticSemaphore_t xMutexBuffer;

void vATask( void * pvParameters )
{
    // A recursive semaphore cannot be used before it is created. Here a
    // recursive mutex is created using xSemaphoreCreateRecursiveMutexStatic().
    // The address of xMutexBuffer is passed into the function, and will hold
    // the mutexes data structures - so no dynamic memory allocation will be
    // attempted.
```

(continues on next page)

(continued from previous page)

```

xSemaphore = xSemaphoreCreateRecursiveMutexStatic( &xMutexBuffer );

// As no dynamic memory allocation was performed, xSemaphore cannot be NULL,
// so there is no need to check it.
}

```

Return If the recursive mutex was successfully created then a handle to the created recursive mutex is returned. If pxMutexBuffer was NULL then NULL is returned.

Parameters

- pxStaticSemaphore: Must point to a variable of type StaticSemaphore_t, which will then be used to hold the recursive mutex's data structure, removing the need for the memory to be allocated dynamically.

xSemaphoreCreateCounting (uxMaxCount, uxInitialCount)

Creates a new counting semaphore instance, and returns a handle by which the new counting semaphore can be referenced.

In many usage scenarios it is faster and more memory efficient to use a direct to task notification in place of a counting semaphore! <http://www.freertos.org/RTOS-task-notifications.html>

Internally, within the FreeRTOS implementation, counting semaphores use a block of memory, in which the counting semaphore structure is stored. If a counting semaphore is created using xSemaphoreCreateCounting() then the required memory is automatically dynamically allocated inside the xSemaphoreCreateCounting() function. (see <http://www.freertos.org/a00111.html>). If a counting semaphore is created using xSemaphoreCreateCountingStatic() then the application writer can instead optionally provide the memory that will get used by the counting semaphore. xSemaphoreCreateCountingStatic() therefore allows a counting semaphore to be created without using any dynamic memory allocation.

Counting semaphores are typically used for two things:

1) Counting events.

In this usage scenario an event handler will 'give' a semaphore each time an event occurs (incrementing the semaphore count value), and a handler task will 'take' a semaphore each time it processes an event (decrementing the semaphore count value). The count value is therefore the difference between the number of events that have occurred and the number that have been processed. In this case it is desirable for the initial count value to be zero.

2) Resource management.

In this usage scenario the count value indicates the number of resources available. To obtain control of a resource a task must first obtain a semaphore - decrementing the semaphore count value. When the count value reaches zero there are no free resources. When a task finishes with the resource it 'gives' the semaphore back - incrementing the semaphore count value. In this case it is desirable for the initial count value to be equal to the maximum count value, indicating that all resources are free.

Example usage:

```

SemaphoreHandle_t xSemaphore;

void vATask( void * pvParameters )
{
    SemaphoreHandle_t xSemaphore = NULL;

    // Semaphore cannot be used before a call to xSemaphoreCreateCounting().
    // The max value to which the semaphore can count should be 10, and the
    // initial value assigned to the count should be 0.
    xSemaphore = xSemaphoreCreateCounting( 10, 0 );

    if( xSemaphore != NULL )
    {
        // The semaphore was created successfully.
    }
}

```

(continues on next page)

(continued from previous page)

```

    // The semaphore can now be used.
}
}

```

Return Handle to the created semaphore. Null if the semaphore could not be created.

Parameters

- **uxMaxCount**: The maximum count value that can be reached. When the semaphore reaches this value it can no longer be ‘given’.
- **uxInitialCount**: The count value assigned to the semaphore when it is created.

xSemaphoreCreateCountingStatic (uxMaxCount, uxInitialCount, pxSemaphoreBuffer)

Creates a new counting semaphore instance, and returns a handle by which the new counting semaphore can be referenced.

In many usage scenarios it is faster and more memory efficient to use a direct to task notification in place of a counting semaphore! <http://www.freertos.org/RTOS-task-notifications.html>

Internally, within the FreeRTOS implementation, counting semaphores use a block of memory, in which the counting semaphore structure is stored. If a counting semaphore is created using `xSemaphoreCreateCounting()` then the required memory is automatically dynamically allocated inside the `xSemaphoreCreateCounting()` function. (see <http://www.freertos.org/a00111.html>). If a counting semaphore is created using `xSemaphoreCreateCountingStatic()` then the application writer must provide the memory. `xSemaphoreCreateCountingStatic()` therefore allows a counting semaphore to be created without using any dynamic memory allocation.

Counting semaphores are typically used for two things:

1) Counting events.

In this usage scenario an event handler will ‘give’ a semaphore each time an event occurs (incrementing the semaphore count value), and a handler task will ‘take’ a semaphore each time it processes an event (decrementing the semaphore count value). The count value is therefore the difference between the number of events that have occurred and the number that have been processed. In this case it is desirable for the initial count value to be zero.

2) Resource management.

In this usage scenario the count value indicates the number of resources available. To obtain control of a resource a task must first obtain a semaphore - decrementing the semaphore count value. When the count value reaches zero there are no free resources. When a task finishes with the resource it ‘gives’ the semaphore back - incrementing the semaphore count value. In this case it is desirable for the initial count value to be equal to the maximum count value, indicating that all resources are free.

Example usage:

```

SemaphoreHandle_t xSemaphore;
StaticSemaphore_t xSemaphoreBuffer;

void vATask( void * pvParameters )
{
    SemaphoreHandle_t xSemaphore = NULL;

    // Counting semaphore cannot be used before they have been created. Create
    // a counting semaphore using xSemaphoreCreateCountingStatic(). The max
    // value to which the semaphore can count is 10, and the initial value
    // assigned to the count will be 0. The address of xSemaphoreBuffer is
    // passed in and will be used to hold the semaphore structure, so no dynamic
    // memory allocation will be used.
    xSemaphore = xSemaphoreCreateCounting( 10, 0, &xSemaphoreBuffer );

    // No memory allocation was attempted so xSemaphore cannot be NULL, so there
    // is no need to check its value.
}

```


Return If the counting semaphore was successfully created then a handle to the created counting semaphore is returned. If `pxSemaphoreBuffer` was NULL then NULL is returned.

Parameters

- `uxMaxCount`: The maximum count value that can be reached. When the semaphore reaches this value it can no longer be ‘given’.
- `uxInitialCount`: The count value assigned to the semaphore when it is created.
- `pxSemaphoreBuffer`: Must point to a variable of type `StaticSemaphore_t`, which will then be used to hold the semaphore’s data structure, removing the need for the memory to be allocated dynamically.

vSemaphoreDelete (`xSemaphore`)

Delete a semaphore. This function must be used with care. For example, do not delete a mutex type semaphore if the mutex is held by a task.

Parameters

- `xSemaphore`: A handle to the semaphore to be deleted.

xSemaphoreGetMutexHolder (`xSemaphore`)

If `xMutex` is indeed a mutex type semaphore, return the current mutex holder. If `xMutex` is not a mutex type semaphore, or the mutex is available (not held by a task), return NULL.

Note: This is a good way of determining if the calling task is the mutex holder, but not a good way of determining the identity of the mutex holder as the holder may change between the function exiting and the returned value being tested.

uxSemaphoreGetCount (`xSemaphore`)

If the semaphore is a counting semaphore then `uxSemaphoreGetCount()` returns its current count value. If the semaphore is a binary semaphore then `uxSemaphoreGetCount()` returns 1 if the semaphore is available, and 0 if the semaphore is not available.

Type Definitions

```
typedef QueueHandle_t SemaphoreHandle_t
```

Timer API

Header File

- [freertos/include/freertos/timers.h](#)

Functions

TimerHandle_t **xTimerCreate** (`const` char *`const pcTimerName`, `const` TickType_t `xTimerPeriodInTicks`, `const` UBaseType_t `uxAutoReload`, void *`const pvTimerID`, *TimerCallbackFunction_t* `pxCallbackFunction`)

Creates a new software timer instance, and returns a handle by which the created software timer can be referenced.

Internally, within the FreeRTOS implementation, software timers use a block of memory, in which the timer data structure is stored. If a software timer is created using `xTimerCreate()` then the required memory is automatically dynamically allocated inside the `xTimerCreate()` function. (see <http://www.freertos.org/a00111.html>). If a software timer is created using `xTimerCreateStatic()` then the application writer must provide the memory that will get used by the software timer. `xTimerCreateStatic()` therefore allows a software timer to be created without using any dynamic memory allocation.

Timers are created in the dormant state. The `xTimerStart()`, `xTimerReset()`, `xTimerStartFromISR()`, `xTimerResetFromISR()`, `xTimerChangePeriod()` and `xTimerChangePeriodFromISR()` API functions can all be used to transition a timer into the active state.

Example usage:


```

#define NUM_TIMERS 5

// An array to hold handles to the created timers.
TimerHandle_t xTimers[ NUM_TIMERS ];

// An array to hold a count of the number of times each timer expires.
int32_t lExpireCounters[ NUM_TIMERS ] = { 0 };

// Define a callback function that will be used by multiple timer instances.
// The callback function does nothing but count the number of times the
// associated timer expires, and stop the timer once the timer has expired
// 10 times.
void vTimerCallback( TimerHandle_t pxTimer )
{
    int32_t lArrayIndex;
    const int32_t xMaxExpiryCountBeforeStopping = 10;

    // Optionally do something if the pxTimer parameter is NULL.
    configASSERT( pxTimer );

    // Which timer expired?
    lArrayIndex = ( int32_t ) pvTimerGetTimerID( pxTimer );

    // Increment the number of times that pxTimer has expired.
    lExpireCounters[ lArrayIndex ] += 1;

    // If the timer has expired 10 times then stop it from running.
    if( lExpireCounters[ lArrayIndex ] == xMaxExpiryCountBeforeStopping )
    {
        // Do not use a block time if calling a timer API function from a
        // timer callback function, as doing so could cause a deadlock!
        xTimerStop( pxTimer, 0 );
    }
}

void main( void )
{
    int32_t x;

    // Create then start some timers. Starting the timers before the scheduler
    // has been started means the timers will start running immediately that
    // the scheduler starts.
    for( x = 0; x < NUM_TIMERS; x++ )
    {
        xTimers[ x ] = xTimerCreate( "Timer", // Just a text name,
        ↪not used by the kernel.
        ( 100 * x ), // The timer period in
        ↪ticks.
        pdTRUE, // The timers will auto-
        ↪reload themselves when they expire.
        ( void * ) x, // Assign each timer a
        ↪unique id equal to its array index.
        vTimerCallback // Each timer calls the
        ↪same callback when it expires.
        );

        if( xTimers[ x ] == NULL )
        {
            // The timer was not created.
        }
        else
        {

```

(continues on next page)

(continued from previous page)

```

// Start the timer. No block time is specified, and even if one
↳was
// it would be ignored because the scheduler has not yet been
// started.
if( xTimerStart( xTimers[ x ], 0 ) != pdPASS )
{
    // The timer could not be set into the Active state.
}
}

// ...
// Create tasks here.
// ...

// Starting the scheduler will start the timers running as they have
↳already
// been set into the active state.
vTaskStartScheduler();

// Should not reach here.
for( ;; );
}

```

Return If the timer is successfully created then a handle to the newly created timer is returned. If the timer cannot be created (because either there is insufficient FreeRTOS heap remaining to allocate the timer structures, or the timer period was set to 0) then NULL is returned.

Parameters

- **pcTimerName**: A text name that is assigned to the timer. This is done purely to assist debugging. The kernel itself only ever references a timer by its handle, and never by its name.
- **xTimerPeriodInTicks**: The timer period. The time is defined in tick periods so the constant `portTICK_PERIOD_MS` can be used to convert a time that has been specified in milliseconds. For example, if the timer must expire after 100 ticks, then `xTimerPeriodInTicks` should be set to 100. Alternatively, if the timer must expire after 500ms, then `xPeriod` can be set to $(500 / \text{portTICK_PERIOD_MS})$ provided `configTICK_RATE_HZ` is less than or equal to 1000.
- **uxAutoReload**: If `uxAutoReload` is set to `pdTRUE` then the timer will expire repeatedly with a frequency set by the `xTimerPeriodInTicks` parameter. If `uxAutoReload` is set to `pdFALSE` then the timer will be a one-shot timer and enter the dormant state after it expires.
- **pvTimerID**: An identifier that is assigned to the timer being created. Typically this would be used in the timer callback function to identify which timer expired when the same callback function is assigned to more than one timer.
- **pxCallbackFunction**: The function to call when the timer expires. Callback functions must have the prototype defined by `TimerCallbackFunction_t`, which is “void vCallbackFunction(TimerHandle_t xTimer);” .

TimerHandle_t xTimerCreateStatic(const char *const pcTimerName, const TickType_t xTimerPeriodInTicks, const UBaseType_t uxAutoReload, void *const pvTimerID, TimerCallbackFunction_t pxCallbackFunction, StaticTimer_t *pxTimerBuffer)

Creates a new software timer instance, and returns a handle by which the created software timer can be referenced.

Internally, within the FreeRTOS implementation, software timers use a block of memory, in which the timer data structure is stored. If a software timer is created using `xTimerCreate()` then the required memory is automatically dynamically allocated inside the `xTimerCreate()` function. (see <http://www.freertos.org/a00111.html>). If a software timer is created using `xTimerCreateStatic()` then the application writer must provide the memory that will get used by the software timer. `xTimerCreateStatic()` therefore allows a software timer to be created without using any dynamic memory allocation.

Timers are created in the dormant state. The `xTimerStart()`, `xTimerReset()`, `xTimerStartFromISR()`, `xTimerResetFromISR()`, `xTimerChangePeriod()` and `xTimerChangePeriodFromISR()` API functions can all be used

to transition a timer into the active state.

Example usage:

```
// The buffer used to hold the software timer's data structure.
static StaticTimer_t xTimerBuffer;

// A variable that will be incremented by the software timer's callback
// function.
UBaseType_t uxVariableToIncrement = 0;

// A software timer callback function that increments a variable passed to
// it when the software timer was created. After the 5th increment the
// callback function stops the software timer.
static void prvTimerCallback( TimerHandle_t xExpiredTimer )
{
    UBaseType_t *puxVariableToIncrement;
    BaseType_t xReturned;

    // Obtain the address of the variable to increment from the timer ID.
    puxVariableToIncrement = ( UBaseType_t * ) pvTimerGetTimerID(
↳xExpiredTimer );

    // Increment the variable to show the timer callback has executed.
    ( *puxVariableToIncrement )++;

    // If this callback has executed the required number of times, stop the
    // timer.
    if( *puxVariableToIncrement == 5 )
    {
        // This is called from a timer callback so must not block.
        xTimerStop( xExpiredTimer, staticDONT_BLOCK );
    }
}

void main( void )
{
    // Create the software time. xTimerCreateStatic() has an extra parameter
    // than the normal xTimerCreate() API function. The parameter is a pointer
    // to the StaticTimer_t structure that will hold the software timer
    // structure. If the parameter is passed as NULL then the structure will
↳be
    // allocated dynamically, just as if xTimerCreate() had been called.
    xTimer = xTimerCreateStatic( "T1", // Text name for the task.
↳
↳Helps debugging only. Not used by FreeRTOS.
    xTimerPeriod, // The period of the timer
↳
↳in ticks.
    pdTRUE, // This is an auto-reload
↳
↳timer.
    ( void * ) &uxVariableToIncrement, // A
↳
↳variable incremented by the software timer's callback function
    prvTimerCallback, // The function to execute
↳
↳when the timer expires.
    &xTimerBuffer ); // The buffer that will
↳
↳hold the software timer structure.

    // The scheduler has not started yet so a block time is not used.
    xReturned = xTimerStart( xTimer, 0 );

    // ...
    // Create tasks here.
    // ...
}
```

(continues on next page)

```

    // Starting the scheduler will start the timers running as they have
    ↪already
    // been set into the active state.
    vTaskStartScheduler();

    // Should not reach here.
    for( ;; );
}

```

Return If the timer is created then a handle to the created timer is returned. If pxTimerBuffer was NULL then NULL is returned.

Parameters

- pcTimerName: A text name that is assigned to the timer. This is done purely to assist debugging. The kernel itself only ever references a timer by its handle, and never by its name.
- xTimerPeriodInTicks: The timer period. The time is defined in tick periods so the constant portTICK_PERIOD_MS can be used to convert a time that has been specified in milliseconds. For example, if the timer must expire after 100 ticks, then xTimerPeriodInTicks should be set to 100. Alternatively, if the timer must expire after 500ms, then xPeriod can be set to (500 / portTICK_PERIOD_MS) provided configTICK_RATE_HZ is less than or equal to 1000.
- uxAutoReload: If uxAutoReload is set to pdTRUE then the timer will expire repeatedly with a frequency set by the xTimerPeriodInTicks parameter. If uxAutoReload is set to pdFALSE then the timer will be a one-shot timer and enter the dormant state after it expires.
- pvTimerID: An identifier that is assigned to the timer being created. Typically this would be used in the timer callback function to identify which timer expired when the same callback function is assigned to more than one timer.
- pxCallbackFunction: The function to call when the timer expires. Callback functions must have the prototype defined by TimerCallbackFunction_t, which is “void vCallbackFunction(TimerHandle_t xTimer);” .
- pxTimerBuffer: Must point to a variable of type StaticTimer_t, which will be then be used to hold the software timer’s data structures, removing the need for the memory to be allocated dynamically.

void ***pvTimerGetTimerID** (*TimerHandle_t xTimer*)

Returns the ID assigned to the timer.

IDs are assigned to timers using the pvTimerID parameter of the call to xTimerCreated() that was used to create the timer.

If the same callback function is assigned to multiple timers then the timer ID can be used within the callback function to identify which timer actually expired.

Example usage:

Return The ID assigned to the timer being queried.

Parameters

- xTimer: The timer being queried.

See the xTimerCreate() API function example usage scenario.

void **vTimerSetTimerID** (*TimerHandle_t xTimer, void *pvNewID*)

Sets the ID assigned to the timer.

IDs are assigned to timers using the pvTimerID parameter of the call to xTimerCreated() that was used to create the timer.

If the same callback function is assigned to multiple timers then the timer ID can be used as time specific (timer local) storage.

Example usage:

Parameters

- xTimer: The timer being updated.

- `pvNewID`: The ID to assign to the timer.

See the `xTimerCreate()` API function example usage scenario.

BaseType_t **xTimerIsTimerActive** (*TimerHandle_t* xTimer)

Queries a timer to see if it is active or dormant.

A timer will be dormant if:

- 1) It has been created but **not** started, **or**
- 2) It **is** an expired one-shot timer that has **not** been restarted.

Timers are created in the dormant state. The `xTimerStart()`, `xTimerReset()`, `xTimerStartFromISR()`, `xTimerResetFromISR()`, `xTimerChangePeriod()` and `xTimerChangePeriodFromISR()` API functions can all be used to transition a timer into the active state.

Example usage:

```
// This function assumes xTimer has already been created.
void vAFunction( TimerHandle_t xTimer )
{
    if( xTimerIsTimerActive( xTimer ) != pdFALSE ) // or more simply and
    →equivalently "if( xTimerIsTimerActive( xTimer ) )"
    {
        // xTimer is active, do something.
    }
    else
    {
        // xTimer is not active, do something else.
    }
}
```

Return `pdFALSE` will be returned if the timer is dormant. A value other than `pdFALSE` will be returned if the timer is active.

Parameters

- `xTimer`: The timer being queried.

TaskHandle_t **xTimerGetTimerDaemonTaskHandle** (void)

`xTimerGetTimerDaemonTaskHandle()` is only available if `INCLUDE_xTimerGetTimerDaemonTaskHandle` is set to 1 in `FreeRTOSConfig.h`.

Simply returns the handle of the timer service/daemon task. It is not valid to call `xTimerGetTimerDaemonTaskHandle()` before the scheduler has been started.

TickType_t **xTimerGetPeriod** (*TimerHandle_t* xTimer)

Returns the period of a timer.

Return The period of the timer in ticks.

Parameters

- `xTimer`: The handle of the timer being queried.

TickType_t **xTimerGetExpiryTime** (*TimerHandle_t* xTimer)

Returns the time in ticks at which the timer will expire. If this is less than the current tick count then the expiry time has overflowed from the current time.

Return If the timer is running then the time in ticks at which the timer will next expire is returned. If the timer is not running then the return value is undefined.

Parameters

- `xTimer`: The handle of the timer being queried.

BaseType_t **xTimerPendFunctionCallFromISR** (*PendedFunction_t* xFunctionToPend, void *pvParameter1, uint32_t ulParameter2, BaseType_t *pxHigherPriorityTaskWoken)

Used from application interrupt service routines to defer the execution of a function to the RTOS daemon task (the timer service task, hence this function is implemented in `timers.c` and is prefixed with 'Timer').

Ideally an interrupt service routine (ISR) is kept as short as possible, but sometimes an ISR either has a lot of processing to do, or needs to perform processing that is not deterministic. In these cases `xTimerPendFunctionCallFromISR()` can be used to defer processing of a function to the RTOS daemon task.

A mechanism is provided that allows the interrupt to return directly to the task that will subsequently execute the pending callback function. This allows the callback function to execute contiguously in time with the interrupt - just as if the callback had executed in the interrupt itself.

Example usage:

```
// The callback function that will execute in the context of the daemon task.
// Note callback functions must all use this same prototype.
void vProcessInterface( void *pvParameter1, uint32_t ulParameter2 )
{
    BaseType_t xInterfaceToService;

    // The interface that requires servicing is passed in the second
    // parameter. The first parameter is not used in this case.
    xInterfaceToService = ( BaseType_t ) ulParameter2;

    // ...Perform the processing here...
}

// An ISR that receives data packets from multiple interfaces
void vAnISR( void )
{
    BaseType_t xInterfaceToService, xHigherPriorityTaskWoken;

    // Query the hardware to determine which interface needs processing.
    xInterfaceToService = prvCheckInterfaces();

    // The actual processing is to be deferred to a task. Request the
    // vProcessInterface() callback function is executed, passing in the
    // number of the interface that needs processing. The interface to
    // service is passed in the second parameter. The first parameter is
    // not used in this case.
    xHigherPriorityTaskWoken = pdFALSE;
    xTimerPendFunctionCallFromISR( vProcessInterface, NULL, ( uint32_t )
    ↪xInterfaceToService, &xHigherPriorityTaskWoken );

    // If xHigherPriorityTaskWoken is now set to pdTRUE then a context
    // switch should be requested. The macro used is port specific and will
    // be either portYIELD_FROM_ISR() or portEND_SWITCHING_ISR() - refer to
    // the documentation page for the port being used.
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Return `pdPASS` is returned if the message was successfully sent to the timer daemon task, otherwise `pdFALSE` is returned.

Parameters

- `xFunctionToPend`: The function to execute from the timer service/ daemon task. The function must conform to the `PendedFunction_t` prototype.
- `pvParameter1`: The value of the callback function's first parameter. The parameter has a void * type to allow it to be used to pass any type. For example, unsigned longs can be cast to a void *, or the void * can be used to point to a structure.
- `ulParameter2`: The value of the callback function's second parameter.
- `pxHigherPriorityTaskWoken`: As mentioned above, calling this function will result in a message being sent to the timer daemon task. If the priority of the timer daemon task (which is set using `configTIMER_TASK_PRIORITY` in `FreeRTOSConfig.h`) is higher than the priority of the currently running task (the task the interrupt interrupted) then `*pxHigherPriorityTaskWoken` will be set to `pdTRUE` within `xTimerPendFunctionCallFromISR()`, indicating that a context switch

should be requested before the interrupt exits. For that reason `*pxHigherPriorityTaskWoken` must be initialised to `pdFALSE`. See the example code below.

`BaseType_t xTimerPendFunctionCall (PendedFunction_t xFunctionToPend, void *pvParameter1, uint32_t ulParameter2, TickType_t xTicksToWait)`

Used to defer the execution of a function to the RTOS daemon task (the timer service task, hence this function is implemented in `timers.c` and is prefixed with 'Timer').

Return `pdPASS` is returned if the message was successfully sent to the timer daemon task, otherwise `pdFALSE` is returned.

Parameters

- `xFunctionToPend`: The function to execute from the timer service/ daemon task. The function must conform to the `PendedFunction_t` prototype.
- `pvParameter1`: The value of the callback function's first parameter. The parameter has a void * type to allow it to be used to pass any type. For example, unsigned longs can be cast to a void *, or the void * can be used to point to a structure.
- `ulParameter2`: The value of the callback function's second parameter.
- `xTicksToWait`: Calling this function will result in a message being sent to the timer daemon task on a queue. `xTicksToWait` is the amount of time the calling task should remain in the Blocked state (so not using any processing time) for space to become available on the timer queue if the queue is found to be full.

`const char *pcTimerGetTimerName (TimerHandle_t xTimer)`

Returns the name that was assigned to a timer when the timer was created.

Return The name assigned to the timer specified by the `xTimer` parameter.

Parameters

- `xTimer`: The handle of the timer being queried.

Macros

`tmrCOMMAND_EXECUTE_CALLBACK_FROM_ISR`

`tmrCOMMAND_EXECUTE_CALLBACK`

`tmrCOMMAND_START_DONT_TRACE`

`tmrCOMMAND_START`

`tmrCOMMAND_RESET`

`tmrCOMMAND_STOP`

`tmrCOMMAND_CHANGE_PERIOD`

`tmrCOMMAND_DELETE`

`tmrFIRST_FROM_ISR_COMMAND`

`tmrCOMMAND_START_FROM_ISR`

`tmrCOMMAND_RESET_FROM_ISR`

`tmrCOMMAND_STOP_FROM_ISR`

`tmrCOMMAND_CHANGE_PERIOD_FROM_ISR`

`xTimerStart (xTimer, xTicksToWait)`

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the `configTIMER_QUEUE_LENGTH` configuration constant.

`xTimerStart()` starts a timer that was previously created using the `xTimerCreate()` API function. If the timer had already been started and was already in the active state, then `xTimerStart()` has equivalent functionality to the `xTimerReset()` API function.

Starting a timer ensures the timer is in the active state. If the timer is not stopped, deleted, or reset in the mean time, the callback function associated with the timer will get called ‘n’ ticks after `xTimerStart()` was called, where ‘n’ is the timers defined period.

It is valid to call `xTimerStart()` before the scheduler has been started, but when this is done the timer will not actually start until the scheduler is started, and the timers expiry time will be relative to when the scheduler is started, not relative to when `xTimerStart()` was called.

The `configUSE_TIMERS` configuration constant must be set to 1 for `xTimerStart()` to be available.

Example usage:

Return `pdFAIL` will be returned if the start command could not be sent to the timer command queue even after `xTicksToWait` ticks had passed. `pdPASS` will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system, although the timers expiry time is relative to when `xTimerStart()` is actually called. The timer service/daemon task priority is set by the `configTIMER_TASK_PRIORITY` configuration constant.

Parameters

- `xTimer`: The handle of the timer being started/restarted.
- `xTicksToWait`: Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the start command to be successfully sent to the timer command queue, should the queue already be full when `xTimerStart()` was called. `xTicksToWait` is ignored if `xTimerStart()` is called before the scheduler is started.

See the `xTimerCreate()` API function example usage scenario.

xTimerStop (`xTimer`, `xTicksToWait`)

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the `configTIMER_QUEUE_LENGTH` configuration constant.

`xTimerStop()` stops a timer that was previously started using either of the `The xTimerStart()`, `xTimerReset()`, `xTimerStartFromISR()`, `xTimerResetFromISR()`, `xTimerChangePeriod()` or `xTimerChangePeriodFromISR()` API functions.

Stopping a timer ensures the timer is not in the active state.

The `configUSE_TIMERS` configuration constant must be set to 1 for `xTimerStop()` to be available.

Example usage:

Return `pdFAIL` will be returned if the stop command could not be sent to the timer command queue even after `xTicksToWait` ticks had passed. `pdPASS` will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the `configTIMER_TASK_PRIORITY` configuration constant.

Parameters

- `xTimer`: The handle of the timer being stopped.
- `xTicksToWait`: Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the stop command to be successfully sent to the timer command queue, should the queue already be full when `xTimerStop()` was called. `xTicksToWait` is ignored if `xTimerStop()` is called before the scheduler is started.

See the `xTimerCreate()` API function example usage scenario.

xTimerChangePeriod (`xTimer`, `xNewPeriod`, `xTicksToWait`)

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the `configTIMER_QUEUE_LENGTH` configuration constant.

`xTimerChangePeriod()` changes the period of a timer that was previously created using the `xTimerCreate()` API function.

xTimerChangePeriod() can be called to change the period of an active or dormant state timer.

The configUSE_TIMERS configuration constant must be set to 1 for xTimerChangePeriod() to be available.

Example usage:

```
// This function assumes xTimer has already been created.  If the timer
// referenced by xTimer is already active when it is called, then the timer
// is deleted.  If the timer referenced by xTimer is not active when it is
// called, then the period of the timer is set to 500ms and the timer is
// started.
void vAFunction( TimerHandle_t xTimer )
{
    if( xTimerIsTimerActive( xTimer ) != pdFALSE ) // or more simply and
    ↪equivalently "if( xTimerIsTimerActive( xTimer ) )"
    {
        // xTimer is already active - delete it.
        xTimerDelete( xTimer );
    }
    else
    {
        // xTimer is not active, change its period to 500ms.  This will also
        // cause the timer to start.  Block for a maximum of 100 ticks if the
        // change period command cannot immediately be sent to the timer
        // command queue.
        if( xTimerChangePeriod( xTimer, 500 / portTICK_PERIOD_MS, 100 ) ==
    ↪pdPASS )
        {
            // The command was successfully sent.
        }
        else
        {
            // The command could not be sent, even after waiting for 100 ticks
            // to pass.  Take appropriate action here.
        }
    }
}
```

Return pdFAIL will be returned if the change period command could not be sent to the timer command queue even after xTicksToWait ticks had passed. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

Parameters

- xTimer: The handle of the timer that is having its period changed.
- xNewPeriod: The new period for xTimer. Timer periods are specified in tick periods, so the constant portTICK_PERIOD_MS can be used to convert a time that has been specified in milliseconds. For example, if the timer must expire after 100 ticks, then xNewPeriod should be set to 100. Alternatively, if the timer must expire after 500ms, then xNewPeriod can be set to (500 / portTICK_PERIOD_MS) provided configTICK_RATE_HZ is less than or equal to 1000.
- xTicksToWait: Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the change period command to be successfully sent to the timer command queue, should the queue already be full when xTimerChangePeriod() was called. xTicksToWait is ignored if xTimerChangePeriod() is called before the scheduler is started.

xTimerDelete(xTimer, xTicksToWait)

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the configTIMER_QUEUE_LENGTH configuration constant.

xTimerDelete() deletes a timer that was previously created using the xTimerCreate() API function.

The `configUSE_TIMERS` configuration constant must be set to 1 for `xTimerDelete()` to be available.

Example usage:

Return `pdFAIL` will be returned if the delete command could not be sent to the timer command queue even after `xTicksToWait` ticks had passed. `pdPASS` will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the `configTIMER_TASK_PRIORITY` configuration constant.

Parameters

- `xTimer`: The handle of the timer being deleted.
- `xTicksToWait`: Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the delete command to be successfully sent to the timer command queue, should the queue already be full when `xTimerDelete()` was called. `xTicksToWait` is ignored if `xTimerDelete()` is called before the scheduler is started.

See the `xTimerChangePeriod()` API function example usage scenario.

xTimerReset (`xTimer`, `xTicksToWait`)

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the `configTIMER_QUEUE_LENGTH` configuration constant.

`xTimerReset()` re-starts a timer that was previously created using the `xTimerCreate()` API function. If the timer had already been started and was already in the active state, then `xTimerReset()` will cause the timer to re-evaluate its expiry time so that it is relative to when `xTimerReset()` was called. If the timer was in the dormant state then `xTimerReset()` has equivalent functionality to the `xTimerStart()` API function.

Resetting a timer ensures the timer is in the active state. If the timer is not stopped, deleted, or reset in the mean time, the callback function associated with the timer will get called ‘n’ ticks after `xTimerReset()` was called, where ‘n’ is the timers defined period.

It is valid to call `xTimerReset()` before the scheduler has been started, but when this is done the timer will not actually start until the scheduler is started, and the timers expiry time will be relative to when the scheduler is started, not relative to when `xTimerReset()` was called.

The `configUSE_TIMERS` configuration constant must be set to 1 for `xTimerReset()` to be available.

Example usage:

```
// When a key is pressed, an LCD back-light is switched on. If 5 seconds pass
// without a key being pressed, then the LCD back-light is switched off. In
// this case, the timer is a one-shot timer.

TimerHandle_t xBacklightTimer = NULL;

// The callback function assigned to the one-shot timer. In this case the
// parameter is not used.
void vBacklightTimerCallback( TimerHandle_t pxTimer )
{
    // The timer expired, therefore 5 seconds must have passed since a key
    // was pressed. Switch off the LCD back-light.
    vSetBacklightState( BACKLIGHT_OFF );
}

// The key press event handler.
void vKeyPressEventHandler( char cKey )
{
    // Ensure the LCD back-light is on, then reset the timer that is
    // responsible for turning the back-light off after 5 seconds of
    // key inactivity. Wait 10 ticks for the command to be successfully sent
    // if it cannot be sent immediately.
    vSetBacklightState( BACKLIGHT_ON );
}
```

(continues on next page)

(continued from previous page)

```

    if( xTimerReset( xBacklightTimer, 100 ) != pdPASS )
    {
        // The reset command was not executed successfully. Take appropriate
        // action here.
    }

    // Perform the rest of the key processing here.
}

void main( void )
{
    int32_t x;

    // Create then start the one-shot timer that is responsible for turning
    // the back-light off if no keys are pressed within a 5 second period.
    xBacklightTimer = xTimerCreate( "BacklightTimer",           // Just a text_
    ↪name, not used by the kernel.
                                ( 5000 / portTICK_PERIOD_MS), // The timer_
    ↪period in ticks.
                                pdFALSE,                       // The timer_
    ↪is a one-shot timer.
                                0,                             // The id is_
    ↪not used by the callback so can take any value.
                                vBacklightTimerCallback        // The_
    ↪callback function that switches the LCD back-light off.
                                );

    if( xBacklightTimer == NULL )
    {
        // The timer was not created.
    }
    else
    {
        // Start the timer. No block time is specified, and even if one was
        // it would be ignored because the scheduler has not yet been
        // started.
        if( xTimerStart( xBacklightTimer, 0 ) != pdPASS )
        {
            // The timer could not be set into the Active state.
        }
    }

    // ...
    // Create tasks here.
    // ...

    // Starting the scheduler will start the timer running as it has already
    // been set into the active state.
    xTaskStartScheduler();

    // Should not reach here.
    for( ;; );
}

```

Return pdFAIL will be returned if the reset command could not be sent to the timer command queue even after xTicksToWait ticks had passed. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system, although the timers expiry time is relative to when xTimerStart() is actually called. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

Parameters

- `xTimer`: The handle of the timer being reset/started/restarted.
- `xTicksToWait`: Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the reset command to be successfully sent to the timer command queue, should the queue already be full when `xTimerReset()` was called. `xTicksToWait` is ignored if `xTimerReset()` is called before the scheduler is started.

xTimerStartFromISR (`xTimer`, `pxHigherPriorityTaskWoken`)

A version of `xTimerStart()` that can be called from an interrupt service routine.

Example usage:

```
// This scenario assumes xBacklightTimer has already been created. When a
// key is pressed, an LCD back-light is switched on. If 5 seconds pass
// without a key being pressed, then the LCD back-light is switched off. In
// this case, the timer is a one-shot timer, and unlike the example given for
// the xTimerReset() function, the key press event handler is an interrupt
// service routine.

// The callback function assigned to the one-shot timer. In this case the
// parameter is not used.
void vBacklightTimerCallback( TimerHandle_t pxTimer )
{
    // The timer expired, therefore 5 seconds must have passed since a key
    // was pressed. Switch off the LCD back-light.
    vSetBacklightState( BACKLIGHT_OFF );
}

// The key press interrupt service routine.
void vKeyPressEventInterruptHandler( void )
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    // Ensure the LCD back-light is on, then restart the timer that is
    // responsible for turning the back-light off after 5 seconds of
    // key inactivity. This is an interrupt service routine so can only
    // call FreeRTOS API functions that end in "FromISR".
    vSetBacklightState( BACKLIGHT_ON );

    // xTimerStartFromISR() or xTimerResetFromISR() could be called here
    // as both cause the timer to re-calculate its expiry time.
    // xHigherPriorityTaskWoken was initialised to pdFALSE when it was
    // declared (in this function).
    if( xTimerStartFromISR( xBacklightTimer, &xHigherPriorityTaskWoken ) !=
    ↪pdPASS )
    {
        // The start command was not executed successfully. Take appropriate
        // action here.
    }

    // Perform the rest of the key processing here.

    // If xHigherPriorityTaskWoken equals pdTRUE, then a context switch
    // should be performed. The syntax required to perform a context switch
    // from inside an ISR varies from port to port, and from compiler to
    // compiler. Inspect the demos for the port you are using to find the
    // actual syntax required.
    if( xHigherPriorityTaskWoken != pdFALSE )
    {
        // Call the interrupt safe yield function here (actual function
        // depends on the FreeRTOS port being used).
    }
}
```

Return `pdFAIL` will be returned if the start command could not be sent to the timer command queue. `pdPASS`

will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system, although the timers expiry time is relative to when `xTimerStartFromISR()` is actually called. The timer service/daemon task priority is set by the `configTIMER_TASK_PRIORITY` configuration constant.

Parameters

- `xTimer`: The handle of the timer being started/restarted.
- `pxHigherPriorityTaskWoken`: The timer service/daemon task spends most of its time in the Blocked state, waiting for messages to arrive on the timer command queue. Calling `xTimerStartFromISR()` writes a message to the timer command queue, so has the potential to transition the timer service/daemon task out of the Blocked state. If calling `xTimerStartFromISR()` causes the timer service/daemon task to leave the Blocked state, and the timer service/ daemon task has a priority equal to or greater than the currently executing task (the task that was interrupted), then `*pxHigherPriorityTaskWoken` will get set to `pdTRUE` internally within the `xTimerStartFromISR()` function. If `xTimerStartFromISR()` sets this value to `pdTRUE` then a context switch should be performed before the interrupt exits.

`xTimerStopFromISR` (`xTimer`, `pxHigherPriorityTaskWoken`)

A version of `xTimerStop()` that can be called from an interrupt service routine.

Example usage:

```
// This scenario assumes xTimer has already been created and started. When
// an interrupt occurs, the timer should be simply stopped.

// The interrupt service routine that stops the timer.
void vAnExampleInterruptServiceRoutine( void )
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    // The interrupt has occurred - simply stop the timer.
    // xHigherPriorityTaskWoken was set to pdFALSE where it was defined
    // (within this function). As this is an interrupt service routine, only
    // FreeRTOS API functions that end in "FromISR" can be used.
    if( xTimerStopFromISR( xTimer, &xHigherPriorityTaskWoken ) != pdPASS )
    {
        // The stop command was not executed successfully. Take appropriate
        // action here.
    }

    // If xHigherPriorityTaskWoken equals pdTRUE, then a context switch
    // should be performed. The syntax required to perform a context switch
    // from inside an ISR varies from port to port, and from compiler to
    // compiler. Inspect the demos for the port you are using to find the
    // actual syntax required.
    if( xHigherPriorityTaskWoken != pdFALSE )
    {
        // Call the interrupt safe yield function here (actual function
        // depends on the FreeRTOS port being used).
    }
}
```

Return `pdFAIL` will be returned if the stop command could not be sent to the timer command queue. `pdPASS` will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the `configTIMER_TASK_PRIORITY` configuration constant.

Parameters

- `xTimer`: The handle of the timer being stopped.
- `pxHigherPriorityTaskWoken`: The timer service/daemon task spends most of its time in the Blocked state, waiting for messages to arrive on the timer command queue. Calling `xTimerStopFromISR()` writes a message to the timer command queue, so has the potential to transition

the timer service/daemon task out of the Blocked state. If calling `xTimerStopFromISR()` causes the timer service/daemon task to leave the Blocked state, and the timer service/daemon task has a priority equal to or greater than the currently executing task (the task that was interrupted), then `*pxHigherPriorityTaskWoken` will get set to `pdTRUE` internally within the `xTimerStopFromISR()` function. If `xTimerStopFromISR()` sets this value to `pdTRUE` then a context switch should be performed before the interrupt exits.

xTimerChangePeriodFromISR (`xTimer`, `xNewPeriod`, `pxHigherPriorityTaskWoken`)

A version of `xTimerChangePeriod()` that can be called from an interrupt service routine.

Example usage:

```
// This scenario assumes xTimer has already been created and started. When
// an interrupt occurs, the period of xTimer should be changed to 500ms.

// The interrupt service routine that changes the period of xTimer.
void vAnExampleInterruptServiceRoutine( void )
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    // The interrupt has occurred - change the period of xTimer to 500ms.
    // xHigherPriorityTaskWoken was set to pdFALSE where it was defined
    // (within this function). As this is an interrupt service routine, only
    // FreeRTOS API functions that end in "FromISR" can be used.
    if( xTimerChangePeriodFromISR( xTimer, &xHigherPriorityTaskWoken ) !=
    ↪pdPASS )
    {
        // The command to change the timers period was not executed
        // successfully. Take appropriate action here.
    }

    // If xHigherPriorityTaskWoken equals pdTRUE, then a context switch
    // should be performed. The syntax required to perform a context switch
    // from inside an ISR varies from port to port, and from compiler to
    // compiler. Inspect the demos for the port you are using to find the
    // actual syntax required.
    if( xHigherPriorityTaskWoken != pdFALSE )
    {
        // Call the interrupt safe yield function here (actual function
        // depends on the FreeRTOS port being used).
    }
}
```

Return `pdFAIL` will be returned if the command to change the timers period could not be sent to the timer command queue. `pdPASS` will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the `configTIMER_TASK_PRIORITY` configuration constant.

Parameters

- `xTimer`: The handle of the timer that is having its period changed.
- `xNewPeriod`: The new period for `xTimer`. Timer periods are specified in tick periods, so the constant `portTICK_PERIOD_MS` can be used to convert a time that has been specified in milliseconds. For example, if the timer must expire after 100 ticks, then `xNewPeriod` should be set to 100. Alternatively, if the timer must expire after 500ms, then `xNewPeriod` can be set to $(500 / \text{portTICK_PERIOD_MS})$ provided `configTICK_RATE_HZ` is less than or equal to 1000.
- `pxHigherPriorityTaskWoken`: The timer service/daemon task spends most of its time in the Blocked state, waiting for messages to arrive on the timer command queue. Calling `xTimerChangePeriodFromISR()` writes a message to the timer command queue, so has the potential to transition the timer service/daemon task out of the Blocked state. If calling `xTimerChangePeriodFromISR()` causes the timer service/daemon task to leave the Blocked state, and the timer service/daemon task has a priority equal to or greater than the currently executing task (the task that was interrupted), then `*pxHigherPriorityTaskWoken` will get set to `pdTRUE` internally within

the `xTimerChangePeriodFromISR()` function. If `xTimerChangePeriodFromISR()` sets this value to `pdTRUE` then a context switch should be performed before the interrupt exits.

xTimerResetFromISR (xTimer, pxHigherPriorityTaskWoken)

A version of `xTimerReset()` that can be called from an interrupt service routine.

Example usage:

```
// This scenario assumes xBacklightTimer has already been created. When a
// key is pressed, an LCD back-light is switched on. If 5 seconds pass
// without a key being pressed, then the LCD back-light is switched off. In
// this case, the timer is a one-shot timer, and unlike the example given for
// the xTimerReset() function, the key press event handler is an interrupt
// service routine.

// The callback function assigned to the one-shot timer. In this case the
// parameter is not used.
void vBacklightTimerCallback( TimerHandle_t pxTimer )
{
    // The timer expired, therefore 5 seconds must have passed since a key
    // was pressed. Switch off the LCD back-light.
    vSetBacklightState( BACKLIGHT_OFF );
}

// The key press interrupt service routine.
void vKeyPressEventInterruptHandler( void )
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    // Ensure the LCD back-light is on, then reset the timer that is
    // responsible for turning the back-light off after 5 seconds of
    // key inactivity. This is an interrupt service routine so can only
    // call FreeRTOS API functions that end in "FromISR".
    vSetBacklightState( BACKLIGHT_ON );

    // xTimerStartFromISR() or xTimerResetFromISR() could be called here
    // as both cause the timer to re-calculate its expiry time.
    // xHigherPriorityTaskWoken was initialised to pdFALSE when it was
    // declared (in this function).
    if( xTimerResetFromISR( xBacklightTimer, &xHigherPriorityTaskWoken ) !=
    ↪pdPASS )
    {
        // The reset command was not executed successfully. Take appropriate
        // action here.
    }

    // Perform the rest of the key processing here.

    // If xHigherPriorityTaskWoken equals pdTRUE, then a context switch
    // should be performed. The syntax required to perform a context switch
    // from inside an ISR varies from port to port, and from compiler to
    // compiler. Inspect the demos for the port you are using to find the
    // actual syntax required.
    if( xHigherPriorityTaskWoken != pdFALSE )
    {
        // Call the interrupt safe yield function here (actual function
        // depends on the FreeRTOS port being used).
    }
}
```

Return `pdFAIL` will be returned if the reset command could not be sent to the timer command queue. `pdPASS` will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system, although the timers expiry time is relative to when `xTimerResetFromISR()` is actually called.

The timer service/daemon task priority is set by the `configTIMER_TASK_PRIORITY` configuration constant.

Parameters

- `xTimer`: The handle of the timer that is to be started, reset, or restarted.
- `pxHigherPriorityTaskWoken`: The timer service/daemon task spends most of its time in the Blocked state, waiting for messages to arrive on the timer command queue. Calling `xTimerResetFromISR()` writes a message to the timer command queue, so has the potential to transition the timer service/daemon task out of the Blocked state. If calling `xTimerResetFromISR()` causes the timer service/daemon task to leave the Blocked state, and the timer service/ daemon task has a priority equal to or greater than the currently executing task (the task that was interrupted), then `*pxHigherPriorityTaskWoken` will get set to `pdTRUE` internally within the `xTimerResetFromISR()` function. If `xTimerResetFromISR()` sets this value to `pdTRUE` then a context switch should be performed before the interrupt exits.

Type Definitions

typedef void ***TimerHandle_t**

Type by which software timers are referenced. For example, a call to `xTimerCreate()` returns an `TimerHandle_t` variable that can then be used to reference the subject timer in calls to other software timer API functions (for example, `xTimerStart()`, `xTimerReset()`, etc.).

typedef void (***TimerCallbackFunction_t**) (*TimerHandle_t* xTimer)

Defines the prototype to which timer callback functions must conform.

typedef void (***PendedFunction_t**) (void *, uint32_t)

Defines the prototype to which functions used with the `xTimerPendFunctionCallFromISR()` function must conform.

Event Group API

Header File

- [freertos/include/freertos/event_groups.h](#)

Functions

EventGroupHandle_t **xEventGroupCreate** (void)

Create a new event group.

Internally, within the FreeRTOS implementation, event groups use a [small] block of memory, in which the event group's structure is stored. If an event groups is created using `xEventGroupCreate()` then the required memory is automatically dynamically allocated inside the `xEventGroupCreate()` function. (see <http://www.freertos.org/a00111.html>). If an event group is created using `xEventGropuCreateStatic()` then the application writer must instead provide the memory that will get used by the event group. `xEventGroupCreateStatic()` therefore allows an event group to be created without using any dynamic memory allocation.

Although event groups are not related to ticks, for internal implementation reasons the number of bits available for use in an event group is dependent on the `configUSE_16_BIT_TICKS` setting in `FreeRTOSConfig.h`. If `configUSE_16_BIT_TICKS` is 1 then each event group contains 8 usable bits (bit 0 to bit 7). If `configUSE_16_BIT_TICKS` is set to 0 then each event group has 24 usable bits (bit 0 to bit 23). The `EventBits_t` type is used to store event bits within an event group.

Example usage:

```
// Declare a variable to hold the created event group.
EventGroupHandle_t xCreatedEventGroup;

// Attempt to create the event group.
xCreatedEventGroup = xEventGroupCreate();

// Was the event group created successfully?
```

(continues on next page)

(continued from previous page)

```

if( xCreatedEventGroup == NULL )
{
    // The event group was not created because there was insufficient
    // FreeRTOS heap available.
}
else
{
    // The event group was created.
}

```

Return If the event group was created then a handle to the event group is returned. If there was insufficient FreeRTOS heap available to create the event group then NULL is returned. See <http://www.freertos.org/a00111.html>

EventGroupHandle_t **xEventGroupCreateStatic** (*StaticEventGroup_t *pxEventGroupBuffer*)

Create a new event group.

Internally, within the FreeRTOS implementation, event groups use a [small] block of memory, in which the event group's structure is stored. If an event groups is created using `xEventGropuCreate()` then the required memory is automatically dynamically allocated inside the `xEventGroupCreate()` function. (see <http://www.freertos.org/a00111.html>). If an event group is created using `xEventGropuCreateStatic()` then the application writer must instead provide the memory that will get used by the event group. `xEventGroupCreateStatic()` therefore allows an event group to be created without using any dynamic memory allocation.

Although event groups are not related to ticks, for internal implementation reasons the number of bits available for use in an event group is dependent on the `configUSE_16_BIT_TICKS` setting in `FreeRTOSConfig.h`. If `configUSE_16_BIT_TICKS` is 1 then each event group contains 8 usable bits (bit 0 to bit 7). If `configUSE_16_BIT_TICKS` is set to 0 then each event group has 24 usable bits (bit 0 to bit 23). The `EventBits_t` type is used to store event bits within an event group.

Example usage:

```

// StaticEventGroup_t is a publicly accessible structure that has the same
// size and alignment requirements as the real event group structure. It is
// provided as a mechanism for applications to know the size of the event
// group (which is dependent on the architecture and configuration file
// settings) without breaking the strict data hiding policy by exposing the
// real event group internals. This StaticEventGroup_t variable is passed
// into the xSemaphoreCreateEventGroupStatic() function and is used to store
// the event group's data structures
StaticEventGroup_t xEventGroupBuffer;

// Create the event group without dynamically allocating any memory.
xEventGroup = xEventGroupCreateStatic( &xEventGroupBuffer );

```

Return If the event group was created then a handle to the event group is returned. If `pxEventGroupBuffer` was NULL then NULL is returned.

Parameters

- `pxEventGroupBuffer`: `pxEventGroupBuffer` must point to a variable of type `StaticEventGroup_t`, which will be then be used to hold the event group's data structures, removing the need for the memory to be allocated dynamically.

EventBits_t **xEventGroupWaitBits** (*EventGroupHandle_t xEventGroup*, **const** *EventBits_t uxBitsToWaitFor*, **const** *BaseType_t xClearOnExit*, **const** *BaseType_t xWaitForAllBits*, *TickType_t xTicksToWait*)

[Potentially] block to wait for one or more bits to be set within a previously created event group.

This function cannot be called from an interrupt.

Example usage:

```

#define BIT_0    ( 1 << 0 )
#define BIT_4    ( 1 << 4 )

```

(continues on next page)

```

void aFunction( EventGroupHandle_t xEventGroup )
{
EventBits_t uxBits;
const TickType_t xTicksToWait = 100 / portTICK_PERIOD_MS;

    // Wait a maximum of 100ms for either bit 0 or bit 4 to be set within
    // the event group. Clear the bits before exiting.
    uxBits = xEventGroupWaitBits(
        xEventGroup,    // The event group being tested.
        BIT_0 | BIT_4, // The bits within the event group to wait
→for.
        pdTRUE,        // BIT_0 and BIT_4 should be cleared before
→returning.
        pdFALSE,       // Don't wait for both bits, either bit will
→do.
        xTicksToWait ); // Wait a maximum of 100ms for either bit to
→be set.

    if( ( uxBits & ( BIT_0 | BIT_4 ) ) == ( BIT_0 | BIT_4 ) )
    {
        // xEventGroupWaitBits() returned because both bits were set.
    }
    else if( ( uxBits & BIT_0 ) != 0 )
    {
        // xEventGroupWaitBits() returned because just BIT_0 was set.
    }
    else if( ( uxBits & BIT_4 ) != 0 )
    {
        // xEventGroupWaitBits() returned because just BIT_4 was set.
    }
    else
    {
        // xEventGroupWaitBits() returned because xTicksToWait ticks passed
        // without either BIT_0 or BIT_4 becoming set.
    }
}

```

{c}

Return The value of the event group at the time either the bits being waited for became set, or the block time expired. Test the return value to know which bits were set. If `xEventGroupWaitBits()` returned because its timeout expired then not all the bits being waited for will be set. If `xEventGroupWaitBits()` returned because the bits it was waiting for were set then the returned value is the event group value before any bits were automatically cleared in the case that `xClearOnExit` parameter was set to `pdTRUE`.

Parameters

- `xEventGroup`: The event group in which the bits are being tested. The event group must have previously been created using a call to `xEventGroupCreate()`.
- `uxBitsToWaitFor`: A bitwise value that indicates the bit or bits to test inside the event group. For example, to wait for bit 0 and/or bit 2 set `uxBitsToWaitFor` to `0x05`. To wait for bits 0 and/or bit 1 and/or bit 2 set `uxBitsToWaitFor` to `0x07`. Etc.
- `xClearOnExit`: If `xClearOnExit` is set to `pdTRUE` then any bits within `uxBitsToWaitFor` that are set within the event group will be cleared before `xEventGroupWaitBits()` returns if the wait condition was met (if the function returns for a reason other than a timeout). If `xClearOnExit` is set to `pdFALSE` then the bits set in the event group are not altered when the call to `xEventGroupWaitBits()` returns.
- `xWaitForAllBits`: If `xWaitForAllBits` is set to `pdTRUE` then `xEventGroupWaitBits()` will return when either all the bits in `uxBitsToWaitFor` are set or the specified block time expires. If `xWaitForAllBits` is set to `pdFALSE` then `xEventGroupWaitBits()` will return when any one of the bits set in `uxBitsToWaitFor` is set or the specified block time expires. The block time is specified by

the `xTicksToWait` parameter.

- `xTicksToWait`: The maximum amount of time (specified in ‘ticks’) to wait for one/all (depending on the `xWaitForAllBits` value) of the bits specified by `uxBitsToWaitFor` to become set.

***EventBits_t* xEventGroupClearBits** (*EventGroupHandle_t* xEventGroup, **const** *EventBits_t* uxBitsToClear)

Clear bits within an event group. This function cannot be called from an interrupt.

Example usage:

```
#define BIT_0    ( 1 << 0 )
#define BIT_4    ( 1 << 4 )

void aFunction( EventGroupHandle_t xEventGroup )
{
    EventBits_t uxBits;

    // Clear bit 0 and bit 4 in xEventGroup.
    uxBits = xEventGroupClearBits(
                xEventGroup,    // The event group being updated.
                BIT_0 | BIT_4 ); // The bits being cleared.

    if( ( uxBits & ( BIT_0 | BIT_4 ) ) == ( BIT_0 | BIT_4 ) )
    {
        // Both bit 0 and bit 4 were set before xEventGroupClearBits() was
        // called. Both will now be clear (not set).
    }
    else if( ( uxBits & BIT_0 ) != 0 )
    {
        // Bit 0 was set before xEventGroupClearBits() was called. It will
        // now be clear.
    }
    else if( ( uxBits & BIT_4 ) != 0 )
    {
        // Bit 4 was set before xEventGroupClearBits() was called. It will
        // now be clear.
    }
    else
    {
        // Neither bit 0 nor bit 4 were set in the first place.
    }
}
```

Return The value of the event group before the specified bits were cleared.

Parameters

- `xEventGroup`: The event group in which the bits are to be cleared.
- `uxBitsToClear`: A bitwise value that indicates the bit or bits to clear in the event group. For example, to clear bit 3 only, set `uxBitsToClear` to `0x08`. To clear bit 3 and bit 0 set `uxBitsToClear` to `0x09`.

***EventBits_t* xEventGroupSetBits** (*EventGroupHandle_t* xEventGroup, **const** *EventBits_t* uxBitsToSet)

Set bits within an event group. This function cannot be called from an interrupt. `xEventGroupSetBitsFromISR()` is a version that can be called from an interrupt.

Setting bits in an event group will automatically unblock tasks that are blocked waiting for the bits.

Example usage:

```
#define BIT_0    ( 1 << 0 )
#define BIT_4    ( 1 << 4 )

void aFunction( EventGroupHandle_t xEventGroup )
{
    EventBits_t uxBits;
```

(continues on next page)

(continued from previous page)

```

// Set bit 0 and bit 4 in xEventGroup.
uxBits = xEventGroupSetBits(
    xEventGroup, // The event group being updated.
    BIT_0 | BIT_4 ); // The bits being set.

if( ( uxBits & ( BIT_0 | BIT_4 ) ) == ( BIT_0 | BIT_4 ) )
{
    // Both bit 0 and bit 4 remained set when the function returned.
}
else if( ( uxBits & BIT_0 ) != 0 )
{
    // Bit 0 remained set when the function returned, but bit 4 was
    // cleared. It might be that bit 4 was cleared automatically as a
    // task that was waiting for bit 4 was removed from the Blocked
    // state.
}
else if( ( uxBits & BIT_4 ) != 0 )
{
    // Bit 4 remained set when the function returned, but bit 0 was
    // cleared. It might be that bit 0 was cleared automatically as a
    // task that was waiting for bit 0 was removed from the Blocked
    // state.
}
else
{
    // Neither bit 0 nor bit 4 remained set. It might be that a task
    // was waiting for both of the bits to be set, and the bits were
    // cleared as the task left the Blocked state.
}
}

```

{c}

Return The value of the event group at the time the call to `xEventGroupSetBits()` returns. There are two reasons why the returned value might have the bits specified by the `uxBitsToSet` parameter cleared. First, if setting a bit results in a task that was waiting for the bit leaving the blocked state then it is possible the bit will be cleared automatically (see the `xClearBitOnExit` parameter of `xEventGroupWaitBits()`). Second, any unblocked (or otherwise Ready state) task that has a priority above that of the task that called `xEventGroupSetBits()` will execute and may change the event group value before the call to `xEventGroupSetBits()` returns.

Parameters

- `xEventGroup`: The event group in which the bits are to be set.
- `uxBitsToSet`: A bitwise value that indicates the bit or bits to set. For example, to set bit 3 only, set `uxBitsToSet` to `0x08`. To set bit 3 and bit 0 set `uxBitsToSet` to `0x09`.

EventBits_t **xEventGroupSync** (*EventGroupHandle_t* `xEventGroup`, **const** *EventBits_t* `uxBitsToSet`, **const** *EventBits_t* `uxBitsToWaitFor`, *TickType_t* `xTicksToWait`)

Atomically set bits within an event group, then wait for a combination of bits to be set within the same event group. This functionality is typically used to synchronise multiple tasks, where each task has to wait for the other tasks to reach a synchronisation point before proceeding.

This function cannot be used from an interrupt.

The function will return before its block time expires if the bits specified by the `uxBitsToWait` parameter are set, or become set within that time. In this case all the bits specified by `uxBitsToWait` will be automatically cleared before the function returns.

Example usage:

```

// Bits used by the three tasks.
#define TASK_0_BIT    ( 1 << 0 )

```

(continues on next page)

(continued from previous page)

```
#define TASK_1_BIT    ( 1 << 1 )
#define TASK_2_BIT    ( 1 << 2 )

#define ALL_SYNC_BITS ( TASK_0_BIT | TASK_1_BIT | TASK_2_BIT )

// Use an event group to synchronise three tasks. It is assumed this event
// group has already been created elsewhere.
EventGroupHandle_t xEventBits;

void vTask0( void *pvParameters )
{
    EventBits_t uxReturn;
    TickType_t xTicksToWait = 100 / portTICK_PERIOD_MS;

    for( ;; )
    {
        // Perform task functionality here.

        // Set bit 0 in the event flag to note this task has reached the
        // sync point. The other two tasks will set the other two bits defined
        // by ALL_SYNC_BITS. All three tasks have reached the synchronisation
        // point when all the ALL_SYNC_BITS are set. Wait a maximum of 100ms
        // for this to happen.
        uxReturn = xEventGroupSync( xEventBits, TASK_0_BIT, ALL_SYNC_BITS,
        ↪xTicksToWait );

        if( ( uxReturn & ALL_SYNC_BITS ) == ALL_SYNC_BITS )
        {
            // All three tasks reached the synchronisation point before the call
            // to xEventGroupSync() timed out.
        }
    }
}

void vTask1( void *pvParameters )
{
    for( ;; )
    {
        // Perform task functionality here.

        // Set bit 1 in the event flag to note this task has reached the
        // synchronisation point. The other two tasks will set the other two
        // bits defined by ALL_SYNC_BITS. All three tasks have reached the
        // synchronisation point when all the ALL_SYNC_BITS are set. Wait
        // indefinitely for this to happen.
        xEventGroupSync( xEventBits, TASK_1_BIT, ALL_SYNC_BITS, portMAX_DELAY );

        // xEventGroupSync() was called with an indefinite block time, so
        // this task will only reach here if the synchronisation was made by all
        // three tasks, so there is no need to test the return value.
    }
}

void vTask2( void *pvParameters )
{
    for( ;; )
    {
        // Perform task functionality here.

        // Set bit 2 in the event flag to note this task has reached the
        // synchronisation point. The other two tasks will set the other two
```

(continues on next page)

(continued from previous page)

```

// bits defined by ALL_SYNC_BITS. All three tasks have reached the
// synchronisation point when all the ALL_SYNC_BITS are set. Wait
// indefinitely for this to happen.
xEventGroupSync( xEventBits, TASK_2_BIT, ALL_SYNC_BITS, portMAX_DELAY );

// xEventGroupSync() was called with an indefinite block time, so
// this task will only reach here if the synchronisation was made by all
// three tasks, so there is no need to test the return value.
}
}

```

Return The value of the event group at the time either the bits being waited for became set, or the block time expired. Test the return value to know which bits were set. If `xEventGroupSync()` returned because its timeout expired then not all the bits being waited for will be set. If `xEventGroupSync()` returned because all the bits it was waiting for were set then the returned value is the event group value before any bits were automatically cleared.

Parameters

- `xEventGroup`: The event group in which the bits are being tested. The event group must have previously been created using a call to `xEventGroupCreate()`.
- `uxBitsToSet`: The bits to set in the event group before determining if, and possibly waiting for, all the bits specified by the `uxBitsToWait` parameter are set.
- `uxBitsToWaitFor`: A bitwise value that indicates the bit or bits to test inside the event group. For example, to wait for bit 0 and bit 2 set `uxBitsToWaitFor` to 0x05. To wait for bits 0 and bit 1 and bit 2 set `uxBitsToWaitFor` to 0x07. Etc.
- `xTicksToWait`: The maximum amount of time (specified in 'ticks') to wait for all of the bits specified by `uxBitsToWaitFor` to become set.

EventBits_t `xEventGroupGetBitsFromISR` (*EventGroupHandle_t* `xEventGroup`)

A version of `xEventGroupGetBits()` that can be called from an ISR.

Return The event group bits at the time `xEventGroupGetBitsFromISR()` was called.

Parameters

- `xEventGroup`: The event group being queried.

`void vEventGroupDelete` (*EventGroupHandle_t* `xEventGroup`)

Delete an event group that was previously created by a call to `xEventGroupCreate()`. Tasks that are blocked on the event group will be unblocked and obtain 0 as the event group's value.

Parameters

- `xEventGroup`: The event group being deleted.

Macros

`xEventGroupClearBitsFromISR` (`xEventGroup`, `uxBitsToClear`)

A version of `xEventGroupClearBits()` that can be called from an interrupt.

Setting bits in an event group is not a deterministic operation because there are an unknown number of tasks that may be waiting for the bit or bits being set. FreeRTOS does not allow nondeterministic operations to be performed while interrupts are disabled, so protects event groups that are accessed from tasks by suspending the scheduler rather than disabling interrupts. As a result event groups cannot be accessed directly from an interrupt service routine. Therefore `xEventGroupClearBitsFromISR()` sends a message to the timer task to have the clear operation performed in the context of the timer task.

Example usage:

```

#define BIT_0    ( 1 << 0 )
#define BIT_4    ( 1 << 4 )

// An event group which it is assumed has already been created by a call to
// xEventGroupCreate().
EventGroupHandle_t xEventGroup;

```

(continues on next page)

```

void anInterruptHandler( void )
{
    // Clear bit 0 and bit 4 in xEventGroup.
    xResult = xEventGroupClearBitsFromISR(
        xEventGroup,    // The event group being updated.
        BIT_0 | BIT_4 ); // The bits being set.

    if( xResult == pdPASS )
    {
        // The message was posted successfully.
    }
}

```

Return If the request to execute the function was posted successfully then `pdPASS` is returned, otherwise `pdFALSE` is returned. `pdFALSE` will be returned if the timer service queue was full.

Parameters

- `xEventGroup`: The event group in which the bits are to be cleared.
- `uxBitsToClear`: A bitwise value that indicates the bit or bits to clear. For example, to clear bit 3 only, set `uxBitsToClear` to `0x08`. To clear bit 3 and bit 0 set `uxBitsToClear` to `0x09`.

xEventGroupSetBitsFromISR (`xEventGroup`, `uxBitsToSet`, `pxHigherPriorityTaskWoken`)

A version of `xEventGroupSetBits()` that can be called from an interrupt.

Setting bits in an event group is not a deterministic operation because there are an unknown number of tasks that may be waiting for the bit or bits being set. FreeRTOS does not allow nondeterministic operations to be performed in interrupts or from critical sections. Therefore `xEventGroupSetBitFromISR()` sends a message to the timer task to have the set operation performed in the context of the timer task - where a scheduler lock is used in place of a critical section.

Example usage:

```

#define BIT_0    ( 1 << 0 )
#define BIT_4    ( 1 << 4 )

// An event group which it is assumed has already been created by a call to
// xEventGroupCreate().
EventGroupHandle_t xEventGroup;

void anInterruptHandler( void )
{
    BaseType_t xHigherPriorityTaskWoken, xResult;

    // xHigherPriorityTaskWoken must be initialised to pdFALSE.
    xHigherPriorityTaskWoken = pdFALSE;

    // Set bit 0 and bit 4 in xEventGroup.
    xResult = xEventGroupSetBitsFromISR(
        xEventGroup,    // The event group being updated.
        BIT_0 | BIT_4  // The bits being set.
        &xHigherPriorityTaskWoken );

    // Was the message posted successfully?
    if( xResult == pdPASS )
    {
        // If xHigherPriorityTaskWoken is now set to pdTRUE then a context
        // switch should be requested. The macro used is port specific and
        // will be either portYIELD_FROM_ISR() or portEND_SWITCHING_ISR() -
        // refer to the documentation page for the port being used.
        portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
    }
}

```

Return If the request to execute the function was posted successfully then pdPASS is returned, otherwise pdFALSE is returned. pdFALSE will be returned if the timer service queue was full.

Parameters

- `xEventGroup`: The event group in which the bits are to be set.
- `uxBitsToSet`: A bitwise value that indicates the bit or bits to set. For example, to set bit 3 only, set `uxBitsToSet` to 0x08. To set bit 3 and bit 0 set `uxBitsToSet` to 0x09.
- `pxHigherPriorityTaskWoken`: As mentioned above, calling this function will result in a message being sent to the timer daemon task. If the priority of the timer daemon task is higher than the priority of the currently running task (the task the interrupt interrupted) then `*pxHigherPriorityTaskWoken` will be set to pdTRUE by `xEventGroupSetBitsFromISR()`, indicating that a context switch should be requested before the interrupt exits. For that reason `*pxHigherPriorityTaskWoken` must be initialised to pdFALSE. See the example code below.

xEventGroupGetBits (`xEventGroup`)

Returns the current value of the bits in an event group. This function cannot be used from an interrupt.

Return The event group bits at the time `xEventGroupGetBits()` was called.

Parameters

- `xEventGroup`: The event group being queried.

Type Definitions

```
typedef void *EventGroupHandle_t
```

An event group is a collection of bits to which an application can assign a meaning. For example, an application may create an event group to convey the status of various CAN bus related events in which bit 0 might mean “A CAN

message has been received and is ready for processing” , bit 1 might mean “The application has queued a message that is ready for sending onto the CAN network” , and bit 2 might mean “It is time to send a SYNC message onto the CAN network” etc. A task can then test the bit values to see which events are active, and optionally enter the Blocked state to wait for a specified bit or a group of specified bits to be active. To continue the CAN bus example, a CAN controlling task can enter the Blocked state (and therefore not consume any processing time) until either bit 0, bit 1 or bit 2 are active, at which time the bit that was actually active would inform the task which action it had to take (process a received message, send a message, or send a SYNC).

The event groups implementation contains intelligence to avoid race conditions that would otherwise occur were an application to use a simple variable for the same purpose. This is particularly important with respect to when a bit within an event group is to be cleared, and when bits have to be set and then tested atomically - as is the case where event groups are used to create a synchronisation point between multiple tasks (a ‘rendezvous’). `event_groups.h`

Type by which event groups are referenced. For example, a call to `xEventGroupCreate()` returns an `EventGroupHandle_t` variable that can then be used as a parameter to other event group functions.

```
typedef TickType_t EventBits_t
```

2.7.9 FreeRTOS Additions

Overview

ESP-IDF FreeRTOS is based on the Xtensa port of FreeRTOS v8.2.0 with significant modifications for SMP compatibility (see [ESP-IDF FreeRTOS SMP Changes](#)). However various features specific to ESP-IDF FreeRTOS have been added. The features are as follows:

Ring Buffers: Ring buffers were added to provide a form of buffer that could accept entries of arbitrary lengths.

Hooks: ESP-IDF FreeRTOS hooks provides support for registering extra Idle and Tick hooks at run time. Moreover, the hooks can be asymmetric amongst both CPUs.

Ring Buffers

The ESP-IDF FreeRTOS ring buffer is a strictly FIFO buffer that supports arbitrarily sized items. Ring buffers are a more memory efficient alternative to FreeRTOS queues in situations where the size of items is variable. The capacity of a ring buffer is not measured by the number of items it can store, but rather by the amount of memory used for storing items. You may apply for a piece of memory on the ring buffer to send an item, or just use the API to copy your data and send (according to the send API you call). For efficiency reasons, **items are always retrieved from the ring buffer by reference**. As a result, all retrieved items *must also be returned* in order for them to be removed from the ring buffer completely. The ring buffers are split into the three following types:

No-Split buffers will guarantee that an item is stored in contiguous memory and will not attempt to split an item under any circumstances. Use no-split buffers when items must occupy contiguous memory. *Only this buffer type allows you getting the data item address and writing to the item by yourself.*

Allow-Split buffers will allow an item to be split when wrapping around if doing so will allow the item to be stored. Allow-split buffers are more memory efficient than no-split buffers but can return an item in two parts when retrieving.

Byte buffers do not store data as separate items. All data is stored as a sequence of bytes, and any number of bytes and be sent or retrieved each time. Use byte buffers when separate items do not need to be maintained (e.g. a byte stream).

Note: No-split/allow-split buffers will always store items at 32-bit aligned addresses. Therefore when retrieving an item, the item pointer is guaranteed to be 32-bit aligned. This is useful especially when you need to send some data to the DMA.

Note: Each item stored in no-split/allow-split buffers will **require an additional 8 bytes for a header**. Item sizes will also be rounded up to a 32-bit aligned size (multiple of 4 bytes), however the true item size is recorded within the header. The sizes of no-split/allow-split buffers will also be rounded up when created.

Usage The following example demonstrates the usage of `xRingbufferCreate()` and `xRingbufferSend()` to create a ring buffer then send an item to it.

```
#include "freertos/ringbuf.h"
static char tx_item[] = "test_item";

...

//Create ring buffer
RingbufHandle_t buf_handle;
buf_handle = xRingbufferCreate(1028, RINGBUF_TYPE_NOSPLIT);
if (buf_handle == NULL) {
    printf("Failed to create ring buffer\n");
}

//Send an item
UBaseType_t res = xRingbufferSend(buf_handle, tx_item, sizeof(tx_item), pdMS_
↵TO_TICKS(1000));
if (res != pdTRUE) {
    printf("Failed to send item\n");
}
```

The following example demonstrates the usage of `xRingbufferSendAcquire()` and `xRingbufferSendComplete()` instead of `xRingbufferSend()` to apply for the memory on the ring buffer (of type `RINGBUF_TYPE_NOSPLIT`) and then send an item to it. This way adds one more step, but allows getting the address of the memory to write to, and writing to the memory yourself.

```

#include "freertos/ringbuf.h"
#include "soc/lldesc.h"

typedef struct {
    lldesc_t dma_desc;
    uint8_t buf[1];
} dma_item_t;

#define DMA_ITEM_SIZE(N) (sizeof(lldesc_t)+((N)+3)&(~3))

...

//Retrieve space for DMA descriptor and corresponding data buffer
//This has to be done with SendAcquire, or the address may be different when_
↪copy
dma_item_t item;
UBaseType_t res = xRingbufferSendAcquire(buf_handle,
                                         &item, DMA_ITEM_SIZE(buffer_size), pdMS_TO_TICKS(1000));
if (res != pdTRUE) {
    printf("Failed to acquire memory for item\n");
}
item->dma_desc = (lldesc_t) {
    .size = buffer_size,
    .length = buffer_size,
    .eof = 0,
    .owner = 1,
    .buf = &item->buf,
};
//Actually send to the ring buffer for consumer to use
res = xRingbufferSendComplete(buf_handle, &item);
if (res != pdTRUE) {
    printf("Failed to send item\n");
}
}

```

The following example demonstrates retrieving and returning an item from a **no-split ring buffer** using `xRingbufferReceive()` and `vRingbufferReturnItem()`

```

...

//Receive an item from no-split ring buffer
size_t item_size;
char *item = (char *)xRingbufferReceive(buf_handle, &item_size, pdMS_TO_
↪TICKS(1000));

//Check received item
if (item != NULL) {
    //Print item
    for (int i = 0; i < item_size; i++) {
        printf("%c", item[i]);
    }
    printf("\n");
    //Return Item
    vRingbufferReturnItem(buf_handle, (void *)item);
} else {
    //Failed to receive item
    printf("Failed to receive item\n");
}
}

```

The following example demonstrates retrieving and returning an item from an **allow-split ring buffer** using `xRingbufferReceiveSplit()` and `vRingbufferReturnItem()`

```

...

//Receive an item from allow-split ring buffer
size_t item_size1, item_size2;
char *item1, *item2;
 BaseType_t ret = xRingbufferReceiveSplit(buf_handle, (void **)&item1, (void_
↳**) &item2, &item_size1, &item_size2, pdMS_TO_TICKS(1000));

//Check received item
if (ret == pdTRUE && item1 != NULL) {
    for (int i = 0; i < item_size1; i++) {
        printf("%c", item1[i]);
    }
    vRingbufferReturnItem(buf_handle, (void *)item1);
    //Check if item was split
    if (item2 != NULL) {
        for (int i = 0; i < item_size2; i++) {
            printf("%c", item2[i]);
        }
        vRingbufferReturnItem(buf_handle, (void *)item2);
    }
    printf("\n");
} else {
    //Failed to receive item
    printf("Failed to receive item\n");
}

```

The following example demonstrates retrieving and returning an item from a **byte buffer** using *xRingbufferReceiveUpTo()* and *vRingbufferReturnItem()*

```

...

//Receive data from byte buffer
size_t item_size;
char *item = (char *)xRingbufferReceiveUpTo(buf_handle, &item_size, pdMS_TO_
↳TICKS(1000), sizeof(tx_item));

//Check received data
if (item != NULL) {
    //Print item
    for (int i = 0; i < item_size; i++) {
        printf("%c", item[i]);
    }
    printf("\n");
    //Return Item
    vRingbufferReturnItem(buf_handle, (void *)item);
} else {
    //Failed to receive item
    printf("Failed to receive item\n");
}

```

For ISR safe versions of the functions used above, call *xRingbufferSendFromISR()*, *xRingbufferReceiveFromISR()*, *xRingbufferReceiveSplitFromISR()*, *xRingbufferReceiveUpToFromISR()*, and *vRingbufferReturnItemFromISR()*

Sending to Ring Buffer The following diagrams illustrate the differences between no-split/allow-split buffers and byte buffers with regards to sending items/data. The diagrams assume that three items of sizes **18, 3, and 27 bytes** are sent respectively to a **buffer of 128 bytes**.

For no-split/allow-split buffers, a header of 8 bytes precedes every data item. Furthermore, the space occupied by each item is **rounded up to the nearest 32-bit aligned size** in order to maintain overall 32-bit alignment. However

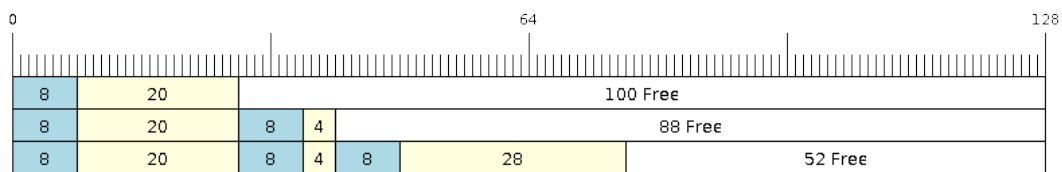


Fig. 26: Sending items to no-split/allow-split ring buffers

the true size of the item is recorded inside the header which will be returned when the item is retrieved. Referring to the diagram above, the 18, 3, and 27 byte items are **rounded up to 20, 4, and 28 bytes** respectively. An 8 byte header is then added in front of each item.

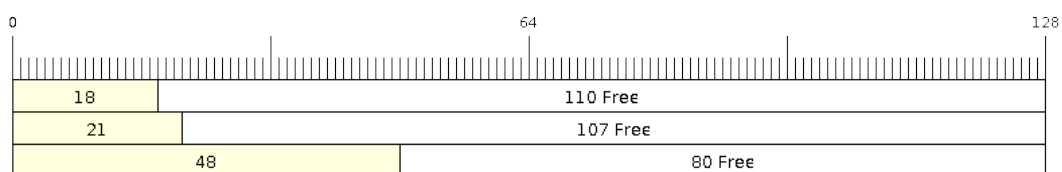


Fig. 27: Sending items to byte buffers

Byte buffers treat data as a sequence of bytes and does not incur any overhead (no headers). As a result, all data sent to a byte buffer is merged into a single item.

Referring to the diagram above, the 18, 3, and 27 byte items are sequentially written to the byte buffer and **merged into a single item of 48 bytes**.

Using SendAcquire and SendComplete Items in no-split buffers are acquired (by SendAcquire) in strict FIFO order and must be sent to the buffer by SendComplete for the data to be accessible by the consumer. Multiple items can be sent or acquired without calling SendComplete, and the items do not necessarily need to be completed in the order they were acquired. However the receiving of data items must occur in FIFO order, therefore not calling SendComplete the earliest acquired item will prevent the subsequent items from being received.

The following diagrams illustrate what will happen when SendAcquire/SendComplete don't happen in the same order. At the beginning, there is already an data item of 16 bytes sent to the ring buffer. Then SendAcquire is called to acquire space of 20, 8, 24 bytes on the ring buffer.

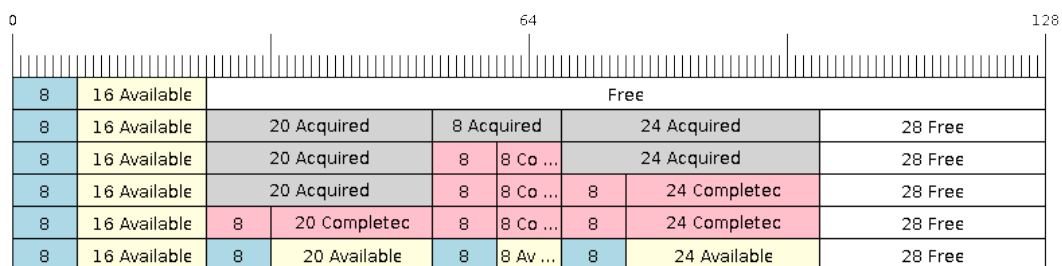


Fig. 28: SendAcquire/SendComplete items in no-split ring buffers

After that, we fill (use) the buffers, and send them to the ring buffer by SendComplete in the order of 8, 24, 20. When

8 bytes and 24 bytes data are sent, the consumer still can only get the 16 bytes data item. Due to the usage if 20 bytes item is not complete, it's not available, nor the following data items.

When the 20 bytes item is finally completed, all the 3 data items can be received now, in the order of 20, 8, 24 bytes, right after the 16 bytes item existing in the buffer at the beginning.

Allow-split/byte buffers do not allow using SendAcquire/SendComplete since acquired buffers are required to be complete (not wrapped).

Wrap around The following diagrams illustrate the differences between no-split, allow-split, and byte buffers when a sent item requires a wrap around. The diagrams assumes a buffer of **128 bytes** with **56 bytes of free space that wraps around** and a sent item of **28 bytes**.

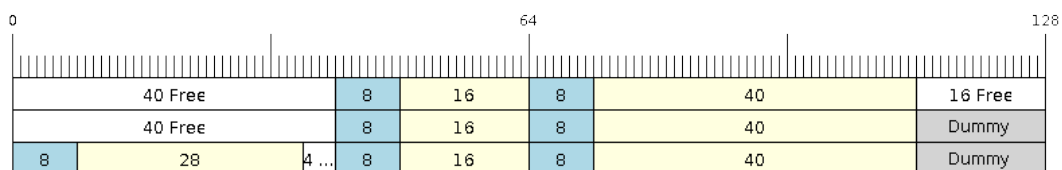


Fig. 29: Wrap around in no-split buffers

No-split buffers will **only store an item in continuous free space and will not split an item under any circumstances**. When the free space at the tail of the buffer is insufficient to completely store the item and its header, the free space at the tail will be **marked as dummy data**. The buffer will then wrap around and store the item in the free space at the head of the buffer.

Referring to the diagram above, the 16 bytes of free space at the tail of the buffer is insufficient to store the 28 byte item. Therefore the 16 bytes is marked as dummy data and the item is written to the free space at the head of the buffer instead.

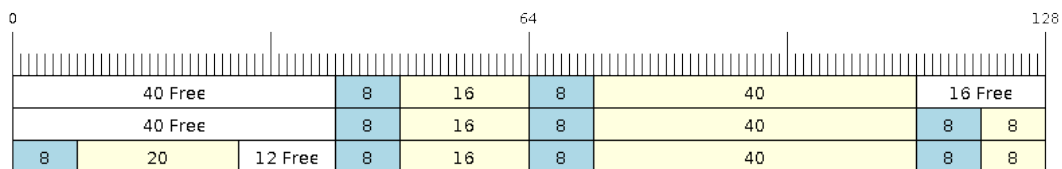


Fig. 30: Wrap around in allow-split buffers

Allow-split buffers will attempt to **split the item into two parts** when the free space at the tail of the buffer is insufficient to store the item data and its header. Both parts of the split item will have their own headers (therefore incurring an extra 8 bytes of overhead).

Referring to the diagram above, the 16 bytes of free space at the tail of the buffer is insufficient to store the 28 byte item. Therefore the item is split into two parts (8 and 20 bytes) and written as two parts to the buffer.

Note: Allow-split buffers treats the both parts of the split item as two separate items, therefore call `xRingbufferReceiveSplit()` instead of `xRingbufferReceive()` to receive both parts of a split item in a thread safe manner.

Byte buffers will **store as much data as possible into the free space at the tail of buffer**. The remaining data will then be stored in the free space at the head of the buffer. No overhead is incurred when wrapping around in byte buffers.

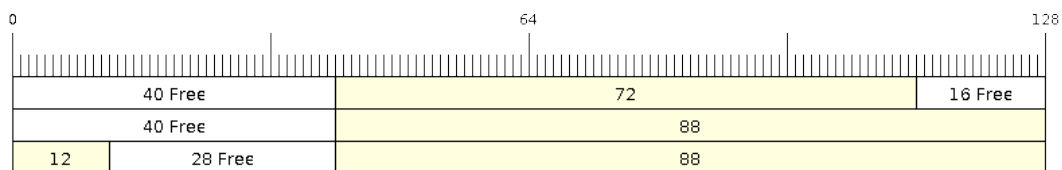


Fig. 31: Wrap around in byte buffers

Referring to the diagram above, the 16 bytes of free space at the tail of the buffer is insufficient to completely store the 28 bytes of data. Therefore the 16 bytes of free space is filled with data, and the remaining 12 bytes are written to the free space at the head of the buffer. The buffer now contains data in two separate continuous parts, and each part continuous will be treated as a separate item by the byte buffer.

Retrieving/Returning The following diagrams illustrates the differences between no-split/allow-split and byte buffers in retrieving and returning data.

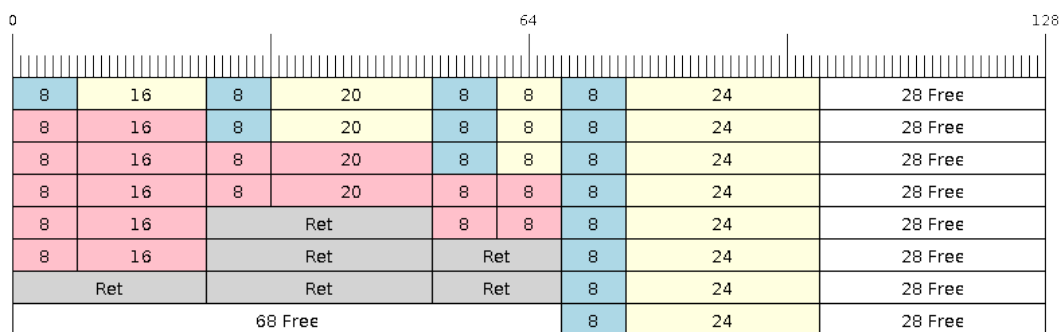


Fig. 32: Retrieving/Returning items in no-split/allow-split ring buffers

Items in no-split/allow-split buffers are **retrieved in strict FIFO order** and **must be returned** for the occupied space to be freed. Multiple items can be retrieved before returning, and the items do not necessarily need to be returned in the order they were retrieved. However the freeing of space must occur in FIFO order, therefore not returning the earliest retrieved item will prevent the space of subsequent items from being freed.

Referring to the diagram above, the **16, 20, and 8 byte items are retrieved in FIFO order**. However the items are not returned in they were retrieved (20, 8, 16). As such, the space is not freed until the first item (16 byte) is returned.

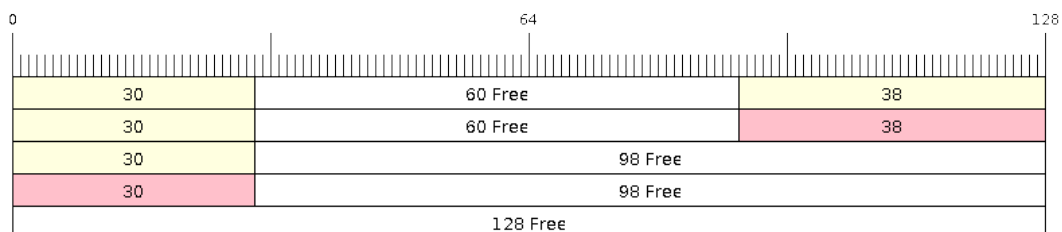


Fig. 33: Retrieving/Returning data in byte buffers

Byte buffers **do not allow multiple retrievals before returning** (every retrieval must be followed by a return

before another retrieval is permitted). When using `xRingbufferReceive()` or `xRingbufferReceiveFromISR()`, all continuous stored data will be retrieved. `xRingbufferReceiveUpTo()` or `xRingbufferReceiveUpToFromISR()` can be used to restrict the maximum number of bytes retrieved. Since every retrieval must be followed by a return, the space will be freed as soon as the data is returned.

Referring to the diagram above, the 38 bytes of continuous stored data at the tail of the buffer is retrieved, returned, and freed. The next call to `xRingbufferReceive()` or `xRingbufferReceiveFromISR()` then wraps around and does the same to the 30 bytes of continuous stored data at the head of the buffer.

Ring Buffers with Queue Sets Ring buffers can be added to FreeRTOS queue sets using `xRingbufferAddToQueueSetRead()` such that every time a ring buffer receives an item or data, the queue set is notified. Once added to a queue set, every attempt to retrieve an item from a ring buffer should be preceded by a call to `xQueueSelectFromSet()`. To check whether the selected queue set member is the ring buffer, call `xRingbufferCanRead()`.

The following example demonstrates queue set usage with ring buffers.

```
#include "freertos/queue.h"
#include "freertos/ringbuf.h"

...

//Create ring buffer and queue set
RingbufHandle_t buf_handle = xRingbufferCreate(1028, RINGBUF_TYPE_NOSPLIT);
QueueSetHandle_t queue_set = xQueueCreateSet(3);

//Add ring buffer to queue set
if (xRingbufferAddToQueueSetRead(buf_handle, queue_set) != pdTRUE) {
    printf("Failed to add to queue set\n");
}

...

//Block on queue set
xQueueSetMemberHandle member = xQueueSelectFromSet(queue_set, pdMS_TO_
↪TICKS(1000));

//Check if member is ring buffer
if (member != NULL && xRingbufferCanRead(buf_handle, member) == pdTRUE) {
    //Member is ring buffer, receive item from ring buffer
    size_t item_size;
    char *item = (char *)xRingbufferReceive(buf_handle, &item_size, 0);

    //Handle item
    ...
} else {
    ...
}
```

Ring Buffers with Static Allocation The `xRingbufferCreateStatic()` can be used to create ring buffers with specific memory requirements (such as a ring buffer being allocated in external RAM). All blocks of memory used by a ring buffer must be manually allocated beforehand then passed to the `xRingbufferCreateStatic()` to be initialized as a ring buffer. These blocks include the following:

- The ring buffer's data structure of type `StaticRingbuffer_t`
- The ring buffer's storage area of size `xBufferSize`. Note that `xBufferSize` must be 32-bit aligned for no-split/allow-split buffers.

The manner in which these blocks are allocated will depend on the users requirements (e.g. all blocks being statically declared, or dynamically allocated with specific capabilities such as external RAM).

Note: The `CONFIG_FREERTOS_SUPPORT_STATIC_ALLOCATION` option must be enabled in `menuconfig` for statically allocated ring buffers to be available.

Note: When deleting a ring buffer created via `xRingbufferCreateStatic()`, the function `vRingbufferDelete()` will not free any of the memory blocks. This must be done manually by the user after `vRingbufferDelete()` is called.

The code snippet below demonstrates a ring buffer being allocated entirely in external RAM.

```
#include "freertos/ringbuf.h"
#include "freertos/semphr.h"
#include "esp_heap_caps.h"

#define BUFFER_SIZE    400    //32-bit aligned size
#define BUFFER_TYPE    RINGBUF_TYPE_NOSPLIT
...

//Allocate ring buffer data structure and storage area into external RAM
StaticRingbuffer_t *buffer_struct = (StaticRingbuffer_t *)heap_caps_
↳malloc(sizeof(StaticRingbuffer_t), MALLOC_CAP_SPIRAM);
uint8_t *buffer_storage = (uint8_t *)heap_caps_malloc(sizeof(uint8_t)*BUFFER_SIZE,↳
↳MALLOC_CAP_SPIRAM);

//Create a ring buffer with manually allocated memory
RingbufHandle_t handle = xRingbufferCreateStatic(BUFFER_SIZE, BUFFER_TYPE, buffer_
↳storage, buffer_struct);

...

//Delete the ring buffer after used
vRingbufferDelete(handle);

//Manually free all blocks of memory
free(buffer_struct);
free(buffer_storage);
```

Ring Buffer API Reference

Note: Ideally, ring buffers can be used with multiple tasks in an SMP fashion where the **highest priority task will always be serviced first**. However due to the usage of binary semaphores in the ring buffer's underlying implementation, priority inversion may occur under very specific circumstances.

The ring buffer governs sending by a binary semaphore which is given whenever space is freed on the ring buffer. The highest priority task waiting to send will repeatedly take the semaphore until sufficient free space becomes available or until it times out. Ideally this should prevent any lower priority tasks from being serviced as the semaphore should always be given to the highest priority task.

However in between iterations of acquiring the semaphore, there is a **gap in the critical section** which may permit another task (on the other core or with an even higher priority) to free some space on the ring buffer and as a result give the semaphore. Therefore the semaphore will be given before the highest priority task can re-acquire the semaphore. This will result in the **semaphore being acquired by the second highest priority task** waiting to send, hence causing priority inversion.

This side effect will not affect ring buffer performance drastically given if the number of tasks using the ring buffer simultaneously is low, and the ring buffer is not operating near maximum capacity.

Header File

- `esp_ringbuf/include/freertos/ringbuf.h`

Functions

RingbufHandle_t **xRingbufferCreate** (*size_t* *xBufferSize*, *RingbufferType_t* *xBufferType*)

Create a ring buffer.

Note *xBufferSize* of no-split/allow-split buffers will be rounded up to the nearest 32-bit aligned size.

Return A handle to the created ring buffer, or NULL in case of error.

Parameters

- [in] *xBufferSize*: Size of the buffer in bytes. Note that items require space for overhead in no-split/allow-split buffers
- [in] *xBufferType*: Type of ring buffer, see documentation.

RingbufHandle_t **xRingbufferCreateNoSplit** (*size_t* *xItemSize*, *size_t* *xItemNum*)

Create a ring buffer of type RINGBUF_TYPE_NOSPLIT for a fixed *item_size*.

This API is similar to `xRingbufferCreate()`, but it will internally allocate additional space for the headers.

Return A *RingbufHandle_t* handle to the created ring buffer, or NULL in case of error.

Parameters

- [in] *xItemSize*: Size of each item to be put into the ring buffer
- [in] *xItemNum*: Maximum number of items the buffer needs to hold simultaneously

RingbufHandle_t **xRingbufferCreateStatic** (*size_t* *xBufferSize*, *RingbufferType_t* *xBufferType*, *uint8_t* **pucRingbufferStorage*, *StaticRingbuffer_t* **pxStaticRingbuffer*)

Create a ring buffer but manually provide the required memory.

Note The `CONFIG_FREERTOS_SUPPORT_STATIC_ALLOCATION` option must be enabled for this to be available

Note *xBufferSize* of no-split/allow-split buffers MUST be 32-bit aligned.

Return A handle to the created ring buffer

Parameters

- [in] *xBufferSize*: Size of the buffer in bytes.
- [in] *xBufferType*: Type of ring buffer, see documentation
- [in] *pucRingbufferStorage*: Pointer to the ring buffer's storage area. Storage area must of the same size as specified by *xBufferSize*
- [in] *pxStaticRingbuffer*: Pointed to a struct of type `StaticRingbuffer_t` which will be used to hold the ring buffer's data structure

BaseType_t **xRingbufferSend** (*RingbufHandle_t* *xRingbuffer*, **const** void **pvItem*, *size_t* *xItemSize*, *TickType_t* *xTicksToWait*)

Insert an item into the ring buffer.

Attempt to insert an item into the ring buffer. This function will block until enough free space is available or until it times out.

Note For no-split/allow-split ring buffers, the actual size of memory that the item will occupy will be rounded up to the nearest 32-bit aligned size. This is done to ensure all items are always stored in 32-bit aligned fashion.

Return

- `pdTRUE` if succeeded
- `pdFALSE` on time-out or when the data is larger than the maximum permissible size of the buffer

Parameters

- [in] *xRingbuffer*: Ring buffer to insert the item into
- [in] *pvItem*: Pointer to data to insert. NULL is allowed if *xItemSize* is 0.
- [in] *xItemSize*: Size of data to insert.
- [in] *xTicksToWait*: Ticks to wait for room in the ring buffer.

BaseType_t **xRingbufferSendFromISR** (*RingbufHandle_t* *xRingbuffer*, **const** void **pvItem*, *size_t* *xItemSize*, *BaseType_t* **pxHigherPriorityTaskWoken*)

Insert an item into the ring buffer in an ISR.

Attempt to insert an item into the ring buffer from an ISR. This function will return immediately if there is insufficient free space in the buffer.

Note For no-split/allow-split ring buffers, the actual size of memory that the item will occupy will be rounded up to the nearest 32-bit aligned size. This is done to ensure all items are always stored in 32-bit aligned fashion.

Return

- pdTRUE if succeeded
- pdFALSE when the ring buffer does not have space.

Parameters

- [in] xRingbuffer: Ring buffer to insert the item into
- [in] pvItem: Pointer to data to insert. NULL is allowed if xItemSize is 0.
- [in] xItemSize: Size of data to insert.
- [out] pxHigherPriorityTaskWoken: Value pointed to will be set to pdTRUE if the function woke up a higher priority task.

BaseType_t **xRingbufferSendAcquire** (*RingbufHandle_t* xRingbuffer, void **ppvItem, size_t xItemSize, TickType_t xTicksToWait)

Acquire memory from the ring buffer to be written to by an external source and to be sent later.

Attempt to allocate buffer for an item to be sent into the ring buffer. This function will block until enough free space is available or until it timesout.

The item, as well as the following items `SendAcquire` or `Send` after it, will not be able to be read from the ring buffer until this item is actually sent into the ring buffer.

Note Only applicable for no-split ring buffers now, the actual size of memory that the item will occupy will be rounded up to the nearest 32-bit aligned size. This is done to ensure all items are always stored in 32-bit aligned fashion.

Return

- pdTRUE if succeeded
- pdFALSE on time-out or when the data is larger than the maximum permissible size of the buffer

Parameters

- [in] xRingbuffer: Ring buffer to allocate the memory
- [out] ppvItem: Double pointer to memory acquired (set to NULL if no memory were retrieved)
- [in] xItemSize: Size of item to acquire.
- [in] xTicksToWait: Ticks to wait for room in the ring buffer.

BaseType_t **xRingbufferSendComplete** (*RingbufHandle_t* xRingbuffer, void *pvItem)

Actually send an item into the ring buffer allocated before by `xRingbufferSendAcquire`.

Note Only applicable for no-split ring buffers. Only call for items allocated by `xRingbufferSendAcquire`.

Return

- pdTRUE if succeeded
- pdFALSE if fail for some reason.

Parameters

- [in] xRingbuffer: Ring buffer to insert the item into
- [in] pvItem: Pointer to item in allocated memory to insert.

void ***xRingbufferReceive** (*RingbufHandle_t* xRingbuffer, size_t *pxItemSize, TickType_t xTicksToWait)

Retrieve an item from the ring buffer.

Attempt to retrieve an item from the ring buffer. This function will block until an item is available or until it times out.

Note A call to `vRingbufferReturnItem()` is required after this to free the item retrieved.

Return

- Pointer to the retrieved item on success; *pxItemSize filled with the length of the item.
- NULL on timeout, *pxItemSize is untouched in that case.

Parameters

- [in] xRingbuffer: Ring buffer to retrieve the item from

- [out] pxItemSize: Pointer to a variable to which the size of the retrieved item will be written.
- [in] xTicksToWait: Ticks to wait for items in the ring buffer.

void ***xRingbufferReceiveFromISR** (*RingbufHandle_t* xRingbuffer, size_t *pxItemSize)

Retrieve an item from the ring buffer in an ISR.

Attempt to retrieve an item from the ring buffer. This function returns immediately if there are no items available for retrieval

Note A call to vRingbufferReturnItemFromISR() is required after this to free the item retrieved.

Note Byte buffers do not allow multiple retrievals before returning an item

Return

- Pointer to the retrieved item on success; *pxItemSize filled with the length of the item.
- NULL when the ring buffer is empty, *pxItemSize is untouched in that case.

Parameters

- [in] xRingbuffer: Ring buffer to retrieve the item from
- [out] pxItemSize: Pointer to a variable to which the size of the retrieved item will be written.

BaseType_t **xRingbufferReceiveSplit** (*RingbufHandle_t* xRingbuffer, void **ppvHeadItem, void **ppvTailItem, size_t *pxHeadItemSize, size_t *pxTailItemSize, TickType_t xTicksToWait)

Retrieve a split item from an allow-split ring buffer.

Attempt to retrieve a split item from an allow-split ring buffer. If the item is not split, only a single item is retrieved. If the item is split, both parts will be retrieved. This function will block until an item is available or until it times out.

Note Call(s) to vRingbufferReturnItem() is required after this to free up the item(s) retrieved.

Note This function should only be called on allow-split buffers

Return

- pdTRUE if an item (split or unsplit) was retrieved
- pdFALSE when no item was retrieved

Parameters

- [in] xRingbuffer: Ring buffer to retrieve the item from
- [out] ppvHeadItem: Double pointer to first part (set to NULL if no items were retrieved)
- [out] ppvTailItem: Double pointer to second part (set to NULL if item is not split)
- [out] pxHeadItemSize: Pointer to size of first part (unmodified if no items were retrieved)
- [out] pxTailItemSize: Pointer to size of second part (unmodified if item is not split)
- [in] xTicksToWait: Ticks to wait for items in the ring buffer.

BaseType_t **xRingbufferReceiveSplitFromISR** (*RingbufHandle_t* xRingbuffer, void **ppvHeadItem, void **ppvTailItem, size_t *pxHeadItemSize, size_t *pxTailItemSize)

Retrieve a split item from an allow-split ring buffer in an ISR.

Attempt to retrieve a split item from an allow-split ring buffer. If the item is not split, only a single item is retrieved. If the item is split, both parts will be retrieved. This function returns immediately if there are no items available for retrieval

Note Calls to vRingbufferReturnItemFromISR() is required after this to free up the item(s) retrieved.

Note This function should only be called on allow-split buffers

Return

- pdTRUE if an item (split or unsplit) was retrieved
- pdFALSE when no item was retrieved

Parameters

- [in] xRingbuffer: Ring buffer to retrieve the item from
- [out] ppvHeadItem: Double pointer to first part (set to NULL if no items were retrieved)
- [out] ppvTailItem: Double pointer to second part (set to NULL if item is not split)
- [out] pxHeadItemSize: Pointer to size of first part (unmodified if no items were retrieved)
- [out] pxTailItemSize: Pointer to size of second part (unmodified if item is not split)

void ***xRingbufferReceiveUpTo** (*RingbufHandle_t* xRingbuffer, size_t *pxItemSize, TickType_t xTicksToWait, size_t xMaxSize)

Retrieve bytes from a byte buffer, specifying the maximum amount of bytes to retrieve.

Attempt to retrieve data from a byte buffer whilst specifying a maximum number of bytes to retrieve. This function will block until there is data available for retrieval or until it times out.

Note A call to `vRingbufferReturnItem()` is required after this to free up the data retrieved.

Note This function should only be called on byte buffers

Note Byte buffers do not allow multiple retrievals before returning an item

Return

- Pointer to the retrieved item on success; `*pxItemSize` filled with the length of the item.
- NULL on timeout, `*pxItemSize` is untouched in that case.

Parameters

- [in] `xRingbuffer`: Ring buffer to retrieve the item from
- [out] `pxItemSize`: Pointer to a variable to which the size of the retrieved item will be written.
- [in] `xTicksToWait`: Ticks to wait for items in the ring buffer.
- [in] `xMaxSize`: Maximum number of bytes to return.

void ***xRingbufferReceiveUpToFromISR** (*RingbufHandle_t* `xRingbuffer`, *size_t* `*pxItemSize`, *size_t* `xMaxSize`)

Retrieve bytes from a byte buffer, specifying the maximum amount of bytes to retrieve. Call this from an ISR.

Attempt to retrieve bytes from a byte buffer whilst specifying a maximum number of bytes to retrieve. This function will return immediately if there is no data available for retrieval.

Note A call to `vRingbufferReturnItemFromISR()` is required after this to free up the data received.

Note This function should only be called on byte buffers

Note Byte buffers do not allow multiple retrievals before returning an item

Return

- Pointer to the retrieved item on success; `*pxItemSize` filled with the length of the item.
- NULL when the ring buffer is empty, `*pxItemSize` is untouched in that case.

Parameters

- [in] `xRingbuffer`: Ring buffer to retrieve the item from
- [out] `pxItemSize`: Pointer to a variable to which the size of the retrieved item will be written.
- [in] `xMaxSize`: Maximum number of bytes to return.

void **vRingbufferReturnItem** (*RingbufHandle_t* `xRingbuffer`, void `*pvItem`)

Return a previously-retrieved item to the ring buffer.

Note If a split item is retrieved, both parts should be returned by calling this function twice

Parameters

- [in] `xRingbuffer`: Ring buffer the item was retrieved from
- [in] `pvItem`: Item that was received earlier

void **vRingbufferReturnItemFromISR** (*RingbufHandle_t* `xRingbuffer`, void `*pvItem`, *BaseType_t* `*pxHigherPriorityTaskWoken`)

Return a previously-retrieved item to the ring buffer from an ISR.

Note If a split item is retrieved, both parts should be returned by calling this function twice

Parameters

- [in] `xRingbuffer`: Ring buffer the item was retrieved from
- [in] `pvItem`: Item that was received earlier
- [out] `pxHigherPriorityTaskWoken`: Value pointed to will be set to `pdTRUE` if the function woke up a higher priority task.

void **vRingbufferDelete** (*RingbufHandle_t* `xRingbuffer`)

Delete a ring buffer.

Note This function will not deallocate any memory if the ring buffer was created using `xRingbufferCreateStatic()`. Deallocation must be done manually by the user.

Parameters

- [in] `xRingbuffer`: Ring buffer to delete

size_t **xRingbufferGetMaxItemSize** (*RingbufHandle_t* `xRingbuffer`)

Get maximum size of an item that can be placed in the ring buffer.

This function returns the maximum size an item can have if it was placed in an empty ring buffer.

Note The max item size for a no-split buffer is limited to $((\text{buffer_size}/2) - \text{header_size})$. This limit is imposed so that an item of max item size can always be sent to the an empty no-split buffer regardless of the internal positions of the buffer's read/write/free pointers.

Return Maximum size, in bytes, of an item that can be placed in a ring buffer.

Parameters

- [in] `xRingbuffer`: Ring buffer to query

`size_t xRingbufferGetCurFreeSize` (*RingbufHandle_t* `xRingbuffer`)

Get current free size available for an item/data in the buffer.

This gives the real time free space available for an item/data in the ring buffer. This represents the maximum size an item/data can have if it was currently sent to the ring buffer.

Warning This API is not thread safe. So, if multiple threads are accessing the same ring buffer, it is the application's responsibility to ensure atomic access to this API and the subsequent Send

Note An empty no-split buffer has a max current free size for an item that is limited to $((\text{buffer_size}/2) - \text{header_size})$. See API reference for `xRingbufferGetMaxItemSize()`.

Return Current free size, in bytes, available for an entry

Parameters

- [in] `xRingbuffer`: Ring buffer to query

`BaseType_t xRingbufferAddToQueueSetRead` (*RingbufHandle_t* `xRingbuffer`, *QueueSetHandle_t* `xQueueSet`)

Add the ring buffer's read semaphore to a queue set.

The ring buffer's read semaphore indicates that data has been written to the ring buffer. This function adds the ring buffer's read semaphore to a queue set.

Return

- `pdTRUE` on success, `pdFALSE` otherwise

Parameters

- [in] `xRingbuffer`: Ring buffer to add to the queue set
- [in] `xQueueSet`: Queue set to add the ring buffer's read semaphore to

`BaseType_t xRingbufferCanRead` (*RingbufHandle_t* `xRingbuffer`, *QueueSetMemberHandle_t* `xMember`)

Check if the selected queue set member is the ring buffer's read semaphore.

This API checks if queue set member returned from `xQueueSelectFromSet()` is the read semaphore of this ring buffer. If so, this indicates the ring buffer has items waiting to be retrieved.

Return

- `pdTRUE` when semaphore belongs to ring buffer
- `pdFALSE` otherwise.

Parameters

- [in] `xRingbuffer`: Ring buffer which should be checked
- [in] `xMember`: Member returned from `xQueueSelectFromSet`

`BaseType_t xRingbufferRemoveFromQueueSetRead` (*RingbufHandle_t* `xRingbuffer`, *QueueSetHandle_t* `xQueueSet`)

Remove the ring buffer's read semaphore from a queue set.

This specifically removes a ring buffer's read semaphore from a queue set. The read semaphore is used to indicate when data has been written to the ring buffer

Return

- `pdTRUE` on success
- `pdFALSE` otherwise

Parameters

- [in] `xRingbuffer`: Ring buffer to remove from the queue set
- [in] `xQueueSet`: Queue set to remove the ring buffer's read semaphore from

`void vRingbufferGetInfo` (*RingbufHandle_t* `xRingbuffer`, `UBaseType_t *uxFree`, `UBaseType_t *uxRead`, `UBaseType_t *uxWrite`, `UBaseType_t *uxAcquire`, `UBaseType_t *uxItemsWaiting`)

Get information about ring buffer status.

Get information of the a ring buffer' s current status such as free/read/write pointer positions, and number of items waiting to be retrieved. Arguments can be set to NULL if they are not required.

Parameters

- [in] `xRingbuffer`: Ring buffer to remove from the queue set
- [out] `uxFree`: Pointer use to store free pointer position
- [out] `uxRead`: Pointer use to store read pointer position
- [out] `uxWrite`: Pointer use to store write pointer position
- [out] `uxAcquire`: Pointer use to store acquire pointer position
- [out] `uxItemsWaiting`: Pointer use to store number of items (bytes for byte buffer) waiting to be retrieved

void **xRingbufferPrintInfo** (*RingbufHandle_t* `xRingbuffer`)

Debugging function to print the internal pointers in the ring buffer.

Parameters

- `xRingbuffer`: Ring buffer to show

Structures

struct xSTATIC_RINGBUFFER

Struct that is equivalent in size to the ring buffer' s data structure.

The contents of this struct are not meant to be used directly. This structure is meant to be used when creating a statically allocated ring buffer where this struct is of the exact size required to store a ring buffer' s control data structure.

Note The `CONFIG_FREERTOS_SUPPORT_STATIC_ALLOCATION` option must be enabled for this structure to be available.

Type Definitions

typedef void *RingbufHandle_t

Type by which ring buffers are referenced. For example, a call to `xRingbufferCreate()` returns a `RingbufHandle_t` variable that can then be used as a parameter to `xRingbufferSend()`, `xRingbufferReceive()`, etc.

typedef struct xSTATIC_RINGBUFFER StaticRingbuffer_t

Struct that is equivalent in size to the ring buffer' s data structure.

The contents of this struct are not meant to be used directly. This structure is meant to be used when creating a statically allocated ring buffer where this struct is of the exact size required to store a ring buffer' s control data structure.

Note The `CONFIG_FREERTOS_SUPPORT_STATIC_ALLOCATION` option must be enabled for this structure to be available.

Enumerations

enum RingbufferType_t

Values:

RINGBUF_TYPE_NOSPLIT = 0

No-split buffers will only store an item in contiguous memory and will never split an item. Each item requires an 8 byte overhead for a header and will always internally occupy a 32-bit aligned size of space.

RINGBUF_TYPE_ALLOWSPLIT

Allow-split buffers will split an item into two parts if necessary in order to store it. Each item requires an 8 byte overhead for a header, splitting incurs an extra header. Each item will always internally occupy a 32-bit aligned size of space.

RINGBUF_TYPE_BYTEBUF

Byte buffers store data as a sequence of bytes and do not maintain separate items, therefore byte buffers have no overhead. All data is stored as a sequence of byte and any number of bytes can be sent or retrieved each time.

RINGBUF_TYPE_MAX

Hooks

FreeRTOS consists of Idle Hooks and Tick Hooks which allow for application specific functionality to be added to the Idle Task and Tick Interrupt. ESP-IDF provides its own Idle and Tick Hook API in addition to the hooks provided by Vanilla FreeRTOS. ESP-IDF hooks have the added benefit of being run time configurable and asymmetrical.

Vanilla FreeRTOS Hooks Idle and Tick Hooks in vanilla FreeRTOS are implemented by the user defining the functions `vApplicationIdleHook()` and `vApplicationTickHook()` respectively somewhere in the application. Vanilla FreeRTOS will run the user defined Idle Hook and Tick Hook on every iteration of the Idle Task and Tick Interrupt respectively.

Vanilla FreeRTOS hooks are referred to as **Legacy Hooks** in ESP-IDF FreeRTOS. To enable legacy hooks, `CONFIG_FREERTOS_LEGACY_HOOKS` should be enabled in *project configuration menu*.

Due to vanilla FreeRTOS being designed for single core, `vApplicationIdleHook()` and `vApplicationTickHook()` can only be defined once. However, the ESP32 is dual core in nature, therefore same Idle Hook and Tick Hook are used for both cores (in other words, the hooks are symmetrical for both cores).

ESP-IDF Idle and Tick Hooks Due to the the dual core nature of the ESP32, it may be necessary for some applications to have separate hooks for each core. Furthermore, it may be necessary for the Idle Tasks or Tick Interrupts to execute multiple hooks that are configurable at run time. Therefore the ESP-IDF provides it' s own hooks API in addition to the legacy hooks provided by Vanilla FreeRTOS.

The ESP-IDF tick/idle hooks are registered at run time, and each tick/idle hook must be registered to a specific CPU. When the idle task runs/tick Interrupt occurs on a particular CPU, the CPU will run each of its registered idle/tick hooks in turn.

Hooks API Reference

Header File

- [esp_common/include/esp_freertos_hooks.h](#)

Functions

`esp_err_t esp_register_freertos_idle_hook_for_cpu(esp_freertos_idle_cb_t new_idle_cb, UBaseType_t cpuid)`

Register a callback to be called from the specified core' s idle hook. The callback should return true if it should be called by the idle hook once per interrupt (or FreeRTOS tick), and return false if it should be called repeatedly as fast as possible by the idle hook.

Warning Idle callbacks MUST NOT, UNDER ANY CIRCUMSTANCES, CALL A FUNCTION THAT MIGHT BLOCK.

Return

- ESP_OK: Callback registered to the specified core' s idle hook
- ESP_ERR_NO_MEM: No more space on the specified core' s idle hook to register callback
- ESP_ERR_INVALID_ARG: cpuid is invalid

Parameters

- [in] `new_idle_cb`: Callback to be called
- [in] `cpuid`: id of the core

`esp_err_t esp_register_freertos_idle_hook(esp_freertos_idle_cb_t new_idle_cb)`

Register a callback to the idle hook of the core that calls this function. The callback should return true if it should be called by the idle hook once per interrupt (or FreeRTOS tick), and return false if it should be called repeatedly as fast as possible by the idle hook.

Warning Idle callbacks MUST NOT, UNDER ANY CIRCUMSTANCES, CALL A FUNCTION THAT MIGHT BLOCK.

Return

- ESP_OK: Callback registered to the calling core' s idle hook
- ESP_ERR_NO_MEM: No more space on the calling core' s idle hook to register callback

Parameters

- [in] *new_idle_cb*: Callback to be called

esp_err_t **esp_register_freertos_tick_hook_for_cpu** (*esp_freertos_tick_cb_t* *new_tick_cb*,
UBaseType_t *cpuid*)

Register a callback to be called from the specified core' s tick hook.

Return

- ESP_OK: Callback registered to specified core' s tick hook
- ESP_ERR_NO_MEM: No more space on the specified core' s tick hook to register the callback
- ESP_ERR_INVALID_ARG: *cpuid* is invalid

Parameters

- [in] *new_tick_cb*: Callback to be called
- [in] *cpuid*: id of the core

esp_err_t **esp_register_freertos_tick_hook** (*esp_freertos_tick_cb_t* *new_tick_cb*)

Register a callback to be called from the calling core' s tick hook.

Return

- ESP_OK: Callback registered to the calling core' s tick hook
- ESP_ERR_NO_MEM: No more space on the calling core' s tick hook to register the callback

Parameters

- [in] *new_tick_cb*: Callback to be called

void **esp_deregister_freertos_idle_hook_for_cpu** (*esp_freertos_idle_cb_t* *old_idle_cb*,
UBaseType_t *cpuid*)

Unregister an idle callback from the idle hook of the specified core.

Parameters

- [in] *old_idle_cb*: Callback to be unregistered
- [in] *cpuid*: id of the core

void **esp_deregister_freertos_idle_hook** (*esp_freertos_idle_cb_t* *old_idle_cb*)

Unregister an idle callback. If the idle callback is registered to the idle hooks of both cores, the idle hook will be unregistered from both cores.

Parameters

- [in] *old_idle_cb*: Callback to be unregistered

void **esp_deregister_freertos_tick_hook_for_cpu** (*esp_freertos_tick_cb_t* *old_tick_cb*,
UBaseType_t *cpuid*)

Unregister a tick callback from the tick hook of the specified core.

Parameters

- [in] *old_tick_cb*: Callback to be unregistered
- [in] *cpuid*: id of the core

void **esp_deregister_freertos_tick_hook** (*esp_freertos_tick_cb_t* *old_tick_cb*)

Unregister a tick callback. If the tick callback is registered to the tick hooks of both cores, the tick hook will be unregistered from both cores.

Parameters

- [in] *old_tick_cb*: Callback to be unregistered

Type Definitions

```
typedef bool (*esp_freertos_idle_cb_t) (void)
```

```
typedef void (*esp_freertos_tick_cb_t) (void)
```

2.7.10 Heap Memory Allocation

Stack and Heap

ESP-IDF applications use the common computer architecture patterns of *stack* (dynamic memory allocated by program control flow) and *heap* (dynamic memory allocated by function calls), as well as statically allocated memory (allocated at compile time).

Because ESP-IDF is a multi-threaded RTOS environment, each RTOS task has its own stack. By default, each of these stacks is allocated from the heap when the task is created. (See `xTaskCreateStatic()` for the alternative where stacks are statically allocated.)

Because ESP32 uses multiple types of RAM, it also contains multiple heaps with different capabilities. A capabilities-based memory allocator allows apps to make heap allocations for different purposes.

For most purposes, the standard libc `malloc()` and `free()` functions can be used for heap allocation without any special consideration.

However, in order to fully make use of all of the memory types and their characteristics, ESP-IDF also has a capabilities-based heap memory allocator. If you want to have memory with certain properties (for example, *DMA-Capable Memory* or executable-memory), you can create an OR-mask of the required capabilities and pass that to `heap_caps_malloc()`.

Memory Capabilities

The ESP32 contains multiple types of RAM:

- DRAM (Data RAM) is memory used to hold data. This is the most common kind of memory accessed as heap.
- IRAM (Instruction RAM) usually holds executable data only. If accessed as generic memory, all accesses must be *32-bit aligned*.
- D/IRAM is RAM which can be used as either Instruction or Data RAM.

For more details on these internal memory types, see *Application memory layout*.

It's also possible to connect external SPI RAM to the ESP32 - *external RAM* can be integrated into the ESP32's memory map using the flash cache, and accessed similarly to DRAM.

DRAM uses capability `MALLOC_CAP_8BIT` (accessible in single byte reads and writes). When calling `malloc()`, the ESP-IDF `malloc()` implementation internally calls `heap_caps_malloc(size, MALLOC_CAP_8BIT)` in order to allocate DRAM that is byte-addressable. To test the free DRAM heap size at runtime, call `cpp:func:heap_caps_get_free_size(MALLOC_CAP_8BIT)`.

Because `malloc` uses the capabilities-based allocation system, memory allocated using `heap_caps_malloc()` can be freed by calling the standard `free()` function.

Available Heap

DRAM At startup, the DRAM heap contains all data memory which is not statically allocated by the app. Reducing statically allocated buffers will increase the amount of available free heap.

To find the amount of statically allocated memory, use the `idf.py size` command.

Note: Due to a technical limitation, the maximum statically allocated DRAM usage is 160KB. The remaining 160KB (for a total of 320KB of DRAM) can only be allocated at runtime as heap.

Note: At runtime, the available heap DRAM may be less than calculated at compile time, because at startup some memory is allocated from the heap before the FreeRTOS scheduler is started (including memory for the stacks of initial FreeRTOS tasks).

IRAM At startup, the IRAM heap contains all instruction memory which is not used by the app executable code.

The `idf.py size` command can be used to find the amount of IRAM used by the app.

D/IRAM Some memory in the ESP32 is available as either DRAM or IRAM. If memory is allocated from a D/IRAM region, the free heap size for both types of memory will decrease.

Heap Sizes At startup, all ESP-IDF apps log a summary of all heap addresses (and sizes) at level Info:

```
I (252) heap_init: Initializing. RAM available for dynamic allocation:
I (259) heap_init: At 3FFAE6E0 len 00001920 (6 KiB): DRAM
I (265) heap_init: At 3FFB2EC8 len 0002D138 (180 KiB): DRAM
I (272) heap_init: At 3FFE0440 len 00003AE0 (14 KiB): D/IRAM
I (278) heap_init: At 3FFE4350 len 0001BCB0 (111 KiB): D/IRAM
I (284) heap_init: At 4008944C len 00016BB4 (90 KiB): IRAM
```

Finding available heap See [Heap Information](#).

Special Capabilities

DMA-Capable Memory Use the `MALLOC_CAP_DMA` flag to allocate memory which is suitable for use with hardware DMA engines (for example SPI and I2S). This capability flag excludes any external PSRAM.

32-Bit Accessible Memory If a certain memory structure is only addressed in 32-bit units, for example an array of ints or pointers, it can be useful to allocate it with the `MALLOC_CAP_32BIT` flag. This also allows the allocator to give out IRAM memory; something which it can't do for a normal `malloc()` call. This can help to use all the available memory in the ESP32.

Memory allocated with `MALLOC_CAP_32BIT` can *only* be accessed via 32-bit reads and writes, any other type of access will generate a fatal `LoadStoreError` exception.

External SPI Memory When [external RAM](#) is enabled, external SPI RAM under 4MiB in size can be allocated using standard `malloc` calls, or via `heap_caps_malloc(MALLOC_CAP_SPIRAM)`, depending on configuration. See [Configuring External RAM](#) for more details.

To use the region above the 4MiB limit, you can use the [himem API](#).

API Reference - Heap Allocation

Header File

- [heap/include/esp_heap_caps.h](#)

Functions

void `*heap_caps_malloc` (size_t *size*, uint32_t *caps*)

Allocate a chunk of memory which has the given capabilities.

Equivalent semantics to `libc malloc()`, for capability-aware memory.

In IDF, `malloc(p)` is equivalent to `heap_caps_malloc(p, MALLOC_CAP_8BIT)`.

Return A pointer to the memory allocated on success, NULL on failure

Parameters

- `size`: Size, in bytes, of the amount of memory to allocate
- `caps`: Bitwise OR of `MALLOC_CAP_*` flags indicating the type of memory to be returned

void **heap_caps_free** (void **ptr*)

Free memory previously allocated via `heap_caps_malloc()` or `heap_caps_realloc()`.

Equivalent semantics to `libc free()`, for capability-aware memory.

In IDF, `free(p)` is equivalent to `heap_caps_free(p)`.

Parameters

- *ptr*: Pointer to memory previously returned from `heap_caps_malloc()` or `heap_caps_realloc()`. Can be NULL.

void ***heap_caps_realloc** (void **ptr*, size_t *size*, int *caps*)

Reallocate memory previously allocated via `heap_caps_malloc()` or `heap_caps_realloc()`.

Equivalent semantics to `libc realloc()`, for capability-aware memory.

In IDF, `realloc(p, s)` is equivalent to `heap_caps_realloc(p, s, MALLOC_CAP_8BIT)`.

'caps' parameter can be different to the capabilities that any original 'ptr' was allocated with. In this way, `realloc` can be used to "move" a buffer if necessary to ensure it meets a new set of capabilities.

Return Pointer to a new buffer of size 'size' with capabilities 'caps', or NULL if allocation failed.

Parameters

- *ptr*: Pointer to previously allocated memory, or NULL for a new allocation.
- *size*: Size of the new buffer requested, or 0 to free the buffer.
- *caps*: Bitwise OR of `MALLOC_CAP_*` flags indicating the type of memory desired for the new allocation.

void ***heap_caps_aligned_alloc** (size_t *alignment*, size_t *size*, int *caps*)

Allocate a aligned chunk of memory which has the given capabilities.

Equivalent semantics to `libc aligned_alloc()`, for capability-aware memory.

Return A pointer to the memory allocated on success, NULL on failure

Note Any memory allocated with `heap_caps_aligned_alloc()` MUST be freed with `heap_caps_aligned_free()` and CANNOT be passed to `free()`

Parameters

- *alignment*: How the pointer received needs to be aligned must be a power of two
- *size*: Size, in bytes, of the amount of memory to allocate
- *caps*: Bitwise OR of `MALLOC_CAP_*` flags indicating the type of memory to be returned

void ***heap_caps_aligned_calloc** (size_t *alignment*, size_t *n*, size_t *size*, uint32_t *caps*)

Allocate a aligned chunk of memory which has the given capabilities. The initialized value in the memory is set to zero.

Return A pointer to the memory allocated on success, NULL on failure

Note Any memory allocated with `heap_caps_aligned_calloc()` MUST be freed with `heap_caps_aligned_free()` and CANNOT be passed to `free()`

Parameters

- *alignment*: How the pointer received needs to be aligned must be a power of two
- *n*: Number of continuing chunks of memory to allocate
- *size*: Size, in bytes, of a chunk of memory to allocate
- *caps*: Bitwise OR of `MALLOC_CAP_*` flags indicating the type of memory to be returned

void **heap_caps_aligned_free** (void **ptr*)

Used to deallocate memory previously allocated with `heap_caps_aligned_alloc`.

Note This function is aimed to deallocate only memory allocated with `heap_caps_aligned_alloc`, memory allocated with `heap_caps_malloc` MUST not be passed to this function

Parameters

- *ptr*: Pointer to the memory allocated

void ***heap_caps_calloc** (size_t *n*, size_t *size*, uint32_t *caps*)

Allocate a chunk of memory which has the given capabilities. The initialized value in the memory is set to zero.

Equivalent semantics to `libc calloc()`, for capability-aware memory.

In IDF, `calloc(p)` is equivalent to `heap_caps_malloc(p, MALLOC_CAP_8BIT)`.

Return A pointer to the memory allocated on success, NULL on failure

Parameters

- `n`: Number of continuing chunks of memory to allocate
- `size`: Size, in bytes, of a chunk of memory to allocate
- `caps`: Bitwise OR of `MALLOC_CAP_*` flags indicating the type of memory to be returned

`size_t heap_caps_get_total_size (uint32_t caps)`

Get the total size of all the regions that have the given capabilities.

This function takes all regions capable of having the given capabilities allocated in them and adds up the total space they have.

Return total size in bytes

Parameters

- `caps`: Bitwise OR of `MALLOC_CAP_*` flags indicating the type of memory

`size_t heap_caps_get_free_size (uint32_t caps)`

Get the total free size of all the regions that have the given capabilities.

This function takes all regions capable of having the given capabilities allocated in them and adds up the free space they have.

Note that because of heap fragmentation it is probably not possible to allocate a single block of memory of this size. Use `heap_caps_get_largest_free_block()` for this purpose.

Return Amount of free bytes in the regions

Parameters

- `caps`: Bitwise OR of `MALLOC_CAP_*` flags indicating the type of memory

`size_t heap_caps_get_minimum_free_size (uint32_t caps)`

Get the total minimum free memory of all regions with the given capabilities.

This adds all the low water marks of the regions capable of delivering the memory with the given capabilities.

Note the result may be less than the global all-time minimum available heap of this kind, as “low water marks” are tracked per-region. Individual regions’ heaps may have reached their “low water marks” at different points in time. However this result still gives a “worst case” indication for all-time minimum free heap.

Return Amount of free bytes in the regions

Parameters

- `caps`: Bitwise OR of `MALLOC_CAP_*` flags indicating the type of memory

`size_t heap_caps_get_largest_free_block (uint32_t caps)`

Get the largest free block of memory able to be allocated with the given capabilities.

Returns the largest value of `s` for which `heap_caps_malloc(s, caps)` will succeed.

Return Size of largest free block in bytes.

Parameters

- `caps`: Bitwise OR of `MALLOC_CAP_*` flags indicating the type of memory

`void heap_caps_get_info (multi_heap_info_t *info, uint32_t caps)`

Get heap info for all regions with the given capabilities.

Calls `multi_heap_info()` on all heaps which share the given capabilities. The information returned is an aggregate across all matching heaps. The meanings of fields are the same as defined for `multi_heap_info_t`, except that `minimum_free_bytes` has the same caveats described in `heap_caps_get_minimum_free_size()`.

Parameters

- `info`: Pointer to a structure which will be filled with relevant heap metadata.
- `caps`: Bitwise OR of `MALLOC_CAP_*` flags indicating the type of memory

`void heap_caps_print_heap_info (uint32_t caps)`

Print a summary of all memory with the given capabilities.

Calls `multi_heap_info` on all heaps which share the given capabilities, and prints a two-line summary for each, then a total summary.

Parameters

- `caps`: Bitwise OR of `MALLOC_CAP_*` flags indicating the type of memory

bool **`heap_caps_check_integrity_all`** (bool *print_errors*)

Check integrity of all heap memory in the system.

Calls `multi_heap_check` on all heaps. Optionally print errors if heaps are corrupt.

Calling this function is equivalent to calling `heap_caps_check_integrity` with the `caps` argument set to `MALLOC_CAP_INVALID`.

Return True if all heaps are valid, False if at least one heap is corrupt.

Parameters

- `print_errors`: Print specific errors if heap corruption is found.

bool **`heap_caps_check_integrity`** (uint32_t *caps*, bool *print_errors*)

Check integrity of all heaps with the given capabilities.

Calls `multi_heap_check` on all heaps which share the given capabilities. Optionally print errors if the heaps are corrupt.

See also `heap_caps_check_integrity_all` to check all heap memory in the system and `heap_caps_check_integrity_addr` to check memory around a single address.

Return True if all heaps are valid, False if at least one heap is corrupt.

Parameters

- `caps`: Bitwise OR of `MALLOC_CAP_*` flags indicating the type of memory
- `print_errors`: Print specific errors if heap corruption is found.

bool **`heap_caps_check_integrity_addr`** (intptr_t *addr*, bool *print_errors*)

Check integrity of heap memory around a given address.

This function can be used to check the integrity of a single region of heap memory, which contains the given address.

This can be useful if debugging heap integrity for corruption at a known address, as it has a lower overhead than checking all heap regions. Note that if the corrupt address moves around between runs (due to timing or other factors) then this approach won't work and you should call `heap_caps_check_integrity` or `heap_caps_check_integrity_all` instead.

Note The entire heap region around the address is checked, not only the adjacent heap blocks.

Return True if the heap containing the specified address is valid, False if at least one heap is corrupt or the address doesn't belong to a heap region.

Parameters

- `addr`: Address in memory. Check for corruption in region containing this address.
- `print_errors`: Print specific errors if heap corruption is found.

void **`heap_caps_malloc_extmem_enable`** (size_t *limit*)

Enable `malloc()` in external memory and set limit below which `malloc()` attempts are placed in internal memory.

When external memory is in use, the allocation strategy is to initially try to satisfy smaller allocation requests with internal memory and larger requests with external memory. This sets the limit between the two, as well as generally enabling allocation in external memory.

Parameters

- `limit`: Limit, in bytes.

void **`*heap_caps_malloc_prefer`** (size_t *size*, size_t *num*, ...)

Allocate a chunk of memory as preference in decreasing order.

Attention The variable parameters are bitwise OR of `MALLOC_CAP_*` flags indicating the type of memory. This API prefers to allocate memory with the first parameter. If failed, allocate memory with the next parameter. It will try in this order until allocating a chunk of memory successfully or fail to allocate memories with any of the parameters.

Return A pointer to the memory allocated on success, NULL on failure

Parameters

- `size`: Size, in bytes, of the amount of memory to allocate
- `num`: Number of variable paramters

void **heap_caps_realloc_prefer** (void **ptr*, size_t *size*, size_t *num*, ...)

Allocate a chunk of memory as preference in decreasing order.

Return Pointer to a new buffer of size ‘`size`’, or NULL if allocation failed.

Parameters

- `ptr`: Pointer to previously allocated memory, or NULL for a new allocation.
- `size`: Size of the new buffer requested, or 0 to free the buffer.
- `num`: Number of variable paramters

void **heap_caps_calloc_prefer** (size_t *n*, size_t *size*, size_t *num*, ...)

Allocate a chunk of memory as preference in decreasing order.

Return A pointer to the memory allocated on success, NULL on failure

Parameters

- `n`: Number of continuing chunks of memory to allocate
- `size`: Size, in bytes, of a chunk of memory to allocate
- `num`: Number of variable paramters

void **heap_caps_dump** (uint32_t *caps*)

Dump the full structure of all heaps with matching capabilities.

Prints a large amount of output to serial (because of locking limitations, the output bypasses stdout/stderr).

For each (variable sized) block in each matching heap, the following output is printed on a single line:

- Block address (the data buffer returned by malloc is 4 bytes after this if heap debugging is set to Basic, or 8 bytes otherwise).
- Data size (the data size may be larger than the size requested by malloc, either due to heap fragmentation or because of heap debugging level).
- Address of next block in the heap.
- If the block is free, the address of the next free block is also printed.

Parameters

- `caps`: Bitwise OR of MALLOC_CAP_* flags indicating the type of memory

void **heap_caps_dump_all** (void)

Dump the full structure of all heaps.

Covers all registered heaps. Prints a large amount of output to serial.

Output is the same as for heap_caps_dump.

size_t **heap_caps_get_allocated_size** (void **ptr*)

Return the size that a particular pointer was allocated with.

Note The app will crash with an assertion failure if the pointer is not valid.

Return Size of the memory allocated at this block.

Parameters

- `ptr`: Pointer to currently allocated heap memory. Must be a pointer value previously returned by heap_caps_malloc, malloc, calloc, etc. and not yet freed.

Macros

MALLOC_CAP_EXEC

Flags to indicate the capabilities of the various memory systems.

Memory must be able to run executable code

MALLOC_CAP_32BIT

Memory must allow for aligned 32-bit data accesses.

MALLOC_CAP_8BIT

Memory must allow for 8/16/...-bit data accesses.

MALLOC_CAP_DMA

Memory must be able to be accessed by DMA.

MALLOC_CAP_PID2

Memory must be mapped to PID2 memory space (PIDs are not currently used)

MALLOC_CAP_PID3

Memory must be mapped to PID3 memory space (PIDs are not currently used)

MALLOC_CAP_PID4

Memory must be mapped to PID4 memory space (PIDs are not currently used)

MALLOC_CAP_PID5

Memory must be mapped to PID5 memory space (PIDs are not currently used)

MALLOC_CAP_PID6

Memory must be mapped to PID6 memory space (PIDs are not currently used)

MALLOC_CAP_PID7

Memory must be mapped to PID7 memory space (PIDs are not currently used)

MALLOC_CAP_SPIRAM

Memory must be in SPI RAM.

MALLOC_CAP_INTERNAL

Memory must be internal; specifically it should not disappear when flash/spiram cache is switched off.

MALLOC_CAP_DEFAULT

Memory can be returned in a non-capability-specific memory allocation (e.g. `malloc()`, `calloc()`) call.

MALLOC_CAP_INVALID

Memory can't be used / list end marker.

Thread Safety Heap functions are thread safe, meaning they can be called from different tasks simultaneously without any limitations.

It is technically possible to call `malloc`, `free`, and related functions from interrupt handler (ISR) context. However this is not recommended, as heap function calls may delay other interrupts. It is strongly recommended to refactor applications so that any buffers used by an ISR are pre-allocated outside of the ISR. Support for calling heap functions from ISRs may be removed in a future update.

Heap Tracing & Debugging

The following features are documented on the [Heap Memory Debugging](#) page:

- [Heap Information](#) (free space, etc.)
- [Heap Corruption Detection](#)
- [Heap Tracing](#) (memory leak detection, monitoring, etc.)

API Reference - Initialisation

Header File

- [heap/include/esp_heap_caps_init.h](#)

Functions

void `heap_caps_init` (void)

Initialize the capability-aware heap allocator.

This is called once in the IDF startup code. Do not call it at other times.

void **heap_caps_enable_nonos_stack_heaps** (void)

Enable heap(s) in memory regions where the startup stacks are located.

On startup, the pro/app CPUs have a certain memory region they use as stack, so we cannot do allocations in the regions these stack frames are. When FreeRTOS is completely started, they do not use that memory anymore and heap(s) there can be enabled.

esp_err_t **heap_caps_add_region** (intptr_t *start*, intptr_t *end*)

Add a region of memory to the collection of heaps at runtime.

Most memory regions are defined in `soc_memory_layout.c` for the SoC, and are registered via `heap_caps_init()`. Some regions can't be used immediately and are later enabled via `heap_caps_enable_nonos_stack_heaps()`.

Call this function to add a region of memory to the heap at some later time.

This function does not consider any of the “reserved” regions or other data in `soc_memory_layout`, caller needs to consider this themselves.

All memory within the region specified by `start` & `end` parameters must be otherwise unused.

The capabilities of the newly registered memory will be determined by the start address, as looked up in the regions specified in `soc_memory_layout.c`.

Use `heap_caps_add_region_with_caps()` to register a region with custom capabilities.

Return `ESP_OK` on success, `ESP_ERR_INVALID_ARG` if a parameter is invalid, `ESP_ERR_NOT_FOUND` if the specified start address doesn't reside in a known region, or any error returned by `heap_caps_add_region_with_caps()`.

Parameters

- `start`: Start address of new region.
- `end`: End address of new region.

esp_err_t **heap_caps_add_region_with_caps** (const uint32_t *caps*[], intptr_t *start*, intptr_t *end*)

Add a region of memory to the collection of heaps at runtime, with custom capabilities.

Similar to `heap_caps_add_region()`, only custom memory capabilities are specified by the caller.

Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_ARG` if a parameter is invalid
- `ESP_ERR_NO_MEM` if no memory to register new heap.
- `ESP_ERR_INVALID_SIZE` if the memory region is too small to fit a heap
- `ESP_FAIL` if region overlaps the start and/or end of an existing region

Parameters

- `caps`: Ordered array of capability masks for the new region, in order of priority. Must have length `SOC_MEMORY_TYPE_NO_PRIOS`. Does not need to remain valid after the call returns.
- `start`: Start address of new region.
- `end`: End address of new region.

Implementation Notes

Knowledge about the regions of memory in the chip comes from the “soc” component, which contains memory layout information for the chip, and the different capabilities of each region. Each region's capabilities are prioritised, so that (for example) dedicated DRAM and IRAM regions will be used for allocations ahead of the more versatile D/IRAM regions.

Each contiguous region of memory contains its own memory heap. The heaps are created using the `multi_heap` functionality. `multi_heap` allows any contiguous region of memory to be used as a heap.

The heap capabilities allocator uses knowledge of the memory regions to initialize each individual heap. Allocation functions in the heap capabilities API will find the most appropriate heap for the allocation (based on desired capabilities, available space, and preferences for each region's use) and then calling `multi_heap_malloc()` or `multi_heap_calloc()` for the heap situated in that particular region.

Calling `free()` involves finding the particular heap corresponding to the freed address, and then calling `multi_heap_free()` on that particular `multi_heap` instance.

API Reference - Multi Heap API

(Note: The multi heap API is used internally by the heap capabilities allocator. Most IDF programs will never need to call this API directly.)

Header File

- [heap/include/multi_heap.h](#)

Functions

void **multi_heap_aligned_alloc** (*multi_heap_handle_t heap*, *size_t size*, *size_t alignment*)
allocate a chunk of memory with specific alignment

Return pointer to the memory allocated, NULL on failure

Parameters

- `heap`: Handle to a registered heap.
- `size`: size in bytes of memory chunk
- `alignment`: how the memory must be aligned

void **multi_heap_malloc** (*multi_heap_handle_t heap*, *size_t size*)
`malloc()` a buffer in a given heap

Semantics are the same as standard `malloc()`, only the returned buffer will be allocated in the specified heap.

Return Pointer to new memory, or NULL if allocation fails.

Parameters

- `heap`: Handle to a registered heap.
- `size`: Size of desired buffer.

void **multi_heap_aligned_free** (*multi_heap_handle_t heap*, *void *p*)
`free()` a buffer aligned in a given heap.

Parameters

- `heap`: Handle to a registered heap.
- `p`: NULL, or a pointer previously returned from `multi_heap_aligned_alloc()` for the same heap.

void **multi_heap_free** (*multi_heap_handle_t heap*, *void *p*)
`free()` a buffer in a given heap.

Semantics are the same as standard `free()`, only the argument `'p'` must be NULL or have been allocated in the specified heap.

Parameters

- `heap`: Handle to a registered heap.
- `p`: NULL, or a pointer previously returned from `multi_heap_malloc()` or `multi_heap_realloc()` for the same heap.

void **multi_heap_realloc** (*multi_heap_handle_t heap*, *void *p*, *size_t size*)
`realloc()` a buffer in a given heap.

Semantics are the same as standard `realloc()`, only the argument `'p'` must be NULL or have been allocated in the specified heap.

Return New buffer of `'size'` containing contents of `'p'`, or NULL if reallocation failed.

Parameters

- `heap`: Handle to a registered heap.
- `p`: NULL, or a pointer previously returned from `multi_heap_malloc()` or `multi_heap_realloc()` for the same heap.
- `size`: Desired new size for buffer.

`size_t multi_heap_get_allocated_size (multi_heap_handle_t heap, void *p)`

Return the size that a particular pointer was allocated with.

Return Size of the memory allocated at this block. May be more than the original size argument, due to padding and minimum block sizes.

Parameters

- `heap`: Handle to a registered heap.
- `p`: Pointer, must have been previously returned from `multi_heap_malloc()` or `multi_heap_realloc()` for the same heap.

`multi_heap_handle_t multi_heap_register (void *start, size_t size)`

Register a new heap for use.

This function initialises a heap at the specified address, and returns a handle for future heap operations.

There is no equivalent function for deregistering a heap - if all blocks in the heap are free, you can immediately start using the memory for other purposes.

Return Handle of a new heap ready for use, or NULL if the heap region was too small to be initialised.

Parameters

- `start`: Start address of the memory to use for a new heap.
- `size`: Size (in bytes) of the new heap.

`void multi_heap_set_lock (multi_heap_handle_t heap, void *lock)`

Associate a private lock pointer with a heap.

The lock argument is supplied to the `MULTI_HEAP_LOCK()` and `MULTI_HEAP_UNLOCK()` macros, defined in `multi_heap_platform.h`.

The lock in question must be recursive.

When the heap is first registered, the associated lock is NULL.

Parameters

- `heap`: Handle to a registered heap.
- `lock`: Optional pointer to a locking structure to associate with this heap.

`void multi_heap_dump (multi_heap_handle_t heap)`

Dump heap information to stdout.

For debugging purposes, this function dumps information about every block in the heap to stdout.

Parameters

- `heap`: Handle to a registered heap.

`bool multi_heap_check (multi_heap_handle_t heap, bool print_errors)`

Check heap integrity.

Walks the heap and checks all heap data structures are valid. If any errors are detected, an error-specific message can be optionally printed to stderr. Print behaviour can be overridden at compile time by defining `MULTI_CHECK_FAIL_PRINTF` in `multi_heap_platform.h`.

Return true if heap is valid, false otherwise.

Parameters

- `heap`: Handle to a registered heap.
- `print_errors`: If true, errors will be printed to stderr.

`size_t multi_heap_free_size (multi_heap_handle_t heap)`

Return free heap size.

Returns the number of bytes available in the heap.

Equivalent to the `total_free_bytes` member returned by `multi_heap_get_heap_info()`.

Note that the heap may be fragmented, so the actual maximum size for a single `malloc()` may be lower. To know this size, see the `largest_free_block` member returned by `multi_heap_get_heap_info()`.

Return Number of free bytes.

Parameters

- `heap`: Handle to a registered heap.

`size_t multi_heap_minimum_free_size (multi_heap_handle_t heap)`

Return the lifetime minimum free heap size.

Equivalent to the `minimum_free_bytes` member returned by `multi_heap_get_info()`.

Returns the lifetime “low water mark” of possible values returned from `multi_free_heap_size()`, for the specified heap.

Return Number of free bytes.

Parameters

- `heap`: Handle to a registered heap.

void `multi_heap_get_info (multi_heap_handle_t heap, multi_heap_info_t *info)`

Return metadata about a given heap.

Fills a `multi_heap_info_t` structure with information about the specified heap.

Parameters

- `heap`: Handle to a registered heap.
- `info`: Pointer to a structure to fill with heap metadata.

Structures

`struct multi_heap_info_t`

Structure to access heap metadata via `multi_heap_get_info`.

Public Members

`size_t total_free_bytes`

Total free bytes in the heap. Equivalent to `multi_free_heap_size()`.

`size_t total_allocated_bytes`

Total bytes allocated to data in the heap.

`size_t largest_free_block`

Size of largest free block in the heap. This is the largest malloc-able size.

`size_t minimum_free_bytes`

Lifetime minimum free heap size. Equivalent to `multi_minimum_free_heap_size()`.

`size_t allocated_blocks`

Number of (variable size) blocks allocated in the heap.

`size_t free_blocks`

Number of (variable size) free blocks in the heap.

`size_t total_blocks`

Total number of (variable size) blocks in the heap.

Type Definitions

`typedef struct multi_heap_info *multi_heap_handle_t`

Opaque handle to a registered heap.

2.7.11 Heap Memory Debugging

Overview

ESP-IDF integrates tools for requesting *heap information*, *detecting heap corruption*, and *tracing memory leaks*. These can help track down memory-related bugs.

For general information about the heap memory allocator, see the *Heap Memory Allocation* page.

Heap Information

To obtain information about the state of the heap:

- `xPortGetFreeHeapSize()` is a FreeRTOS function which returns the number of free bytes in the (data memory) heap. This is equivalent to calling `heap_caps_get_free_size(MALLOC_CAP_8BIT)`.
- `heap_caps_get_free_size()` can also be used to return the current free memory for different memory capabilities.
- `heap_caps_get_largest_free_block()` can be used to return the largest free block in the heap. This is the largest single allocation which is currently possible. Tracking this value and comparing to total free heap allows you to detect heap fragmentation.
- `xPortGetMinimumEverFreeHeapSize()` and the related `heap_caps_get_minimum_free_size()` can be used to track the heap “low water mark” since boot.
- `heap_caps_get_info()` returns a `multi_heap_info_t` structure which contains the information from the above functions, plus some additional heap-specific data (number of allocations, etc.).
- `heap_caps_print_heap_info()` prints a summary to stdout of the information returned by `heap_caps_get_info()`.
- `heap_caps_dump()` and `heap_caps_dump_all()` will output detailed information about the structure of each block in the heap. Note that this can be large amount of output.

Heap Corruption Detection

Heap corruption detection allows you to detect various types of heap memory errors:

- Out of bounds writes & buffer overflow.
- Writes to freed memory.
- Reads from freed or uninitialized memory,

Assertions The heap implementation (`multi_heap.c`, etc.) includes a lot of assertions which will fail if the heap memory is corrupted. To detect heap corruption most effectively, ensure that assertions are enabled in the project configuration menu under `Compiler options` -> `CONFIG_COMPILER_OPTIMIZATION_ASSERTION_LEVEL`.

If a heap integrity assertion fails, a line will be printed like `CORRUPT HEAP: multi_heap.c:225 detected at 0x3ffbb71c`. The memory address which is printed is the address of the heap structure which has corrupt content.

It's also possible to manually check heap integrity by calling `heap_caps_check_integrity_all()` or related functions. This function checks all of requested heap memory for integrity, and can be used even if assertions are disabled. If the integrity check prints an error, it will also contain the address(es) of corrupt heap structures.

Finding Heap Corruption Memory corruption can be one of the hardest classes of bugs to find and fix, as one area of memory can be corrupted from a totally different place. Some tips:

- A crash with a `CORRUPT HEAP:` message will usually include a stack trace, but this stack trace is rarely useful. The crash is the symptom of memory corruption when the system realises the heap is corrupt, but usually the corruption happened elsewhere and earlier in time.
- Increasing the Heap memory debugging *Configuration* level to “Light impact” or “Comprehensive” can give you a more accurate message with the first corrupt memory address.
- Adding regular calls to `heap_caps_check_integrity_all()` or `heap_caps_check_integrity_addr()` in your code will help you pin down the exact time that the corruption happened. You can move these checks around to “close in on” the section of code that corrupted the heap.
- Based on the memory address which is being corrupted, you can use *JTAG debugging* to set a watchpoint on this address and have the CPU halt when it is written to.
- If you don't have JTAG, but you do know roughly when the corruption happens, then you can set a watchpoint in software just beforehand via `esp_set_watchpoint()`. A fatal exception will occur when the watchpoint triggers. For example `esp_set_watchpoint(0, (void *)addr, 4, ESP_WATCHPOINT_STORE`. Note that watchpoints are per-CPU and are set on the current running CPU

only, so if you don't know which CPU is corrupting memory then you will need to call this function on both CPUs.

- For buffer overflows, *heap tracing* in `HEAP_TRACE_ALL` mode lets you see which callers are allocating which addresses from the heap. See *Heap Tracing To Find Heap Corruption* for more details. If you can find the function which allocates memory with an address immediately before the address which is corrupted, this will probably be the function which overflows the buffer.
- Calling `heap_caps_dump()` or `heap_caps_dump_all()` can give an indication of what heap blocks are surrounding the corrupted region and may have overflowed/underflowed/etc.

Configuration Temporarily increasing the heap corruption detection level can give more detailed information about heap corruption errors.

In the project configuration menu, under `Component config` there is a menu `Heap memory debugging`. The setting `CONFIG_HEAP_CORRUPTION_DETECTION` can be set to one of three levels:

Basic (no poisoning) This is the default level. No special heap corruption features are enabled, but provided assertions are enabled (the default configuration) then a heap corruption error will be printed if any of the heap's internal data structures appear overwritten or corrupted. This usually indicates a buffer overrun or out of bounds write.

If assertions are enabled, an assertion will also trigger if a double-free occurs (the same memory is freed twice).

Calling `heap_caps_check_integrity()` in Basic mode will check the integrity of all heap structures, and print errors if any appear to be corrupted.

Light Impact At this level, heap memory is additionally “poisoned” with head and tail “canary bytes” before and after each block which is allocated. If an application writes outside the bounds of allocated buffers, the canary bytes will be corrupted and the integrity check will fail.

The head canary word is `0xABBA1234` (`3412BAAB` in byte order), and the tail canary word is `0xBAAD5678` (`7856ADBA` in byte order).

“Basic” heap corruption checks can also detect most out of bounds writes, but this setting is more precise as even a single byte overrun can be detected. With Basic heap checks, the number of overrun bytes before a failure is detected will depend on the properties of the heap.

Enabling “Light Impact” checking increases memory usage, each individual allocation will use 9 to 12 additional bytes of memory (depending on alignment).

Each time `free()` is called in Light Impact mode, the head and tail canary bytes of the buffer being freed are checked against the expected values.

When `heap_caps_check_integrity()` is called, all allocated blocks of heap memory have their canary bytes checked against the expected values.

In both cases, the check is that the first 4 bytes of an allocated block (before the buffer returned to the user) should be the word `0xABBA1234`. Then the last 4 bytes of the allocated block (after the buffer returned to the user) should be the word `0xBAAD5678`.

Different values usually indicate buffer underrun or overrun, respectively.

Comprehensive This level incorporates the “light impact” detection features plus additional checks for uninitialised-access and use-after-free bugs. In this mode, all freshly allocated memory is filled with the pattern `0xCE`, and all freed memory is filled with the pattern `0xFE`.

Enabling “Comprehensive” detection has a substantial runtime performance impact (as all memory needs to be set to the allocation patterns each time a `malloc/free` completes, and the memory also needs to be checked each time.) However it allows easier detection of memory corruption bugs which are much more subtle to find otherwise. It is recommended to only enable this mode when debugging, not in production.

Crashes in Comprehensive Mode If an application crashes reading/writing an address related to 0xCECECECE in Comprehensive mode, this indicates it has read uninitialized memory. The application should be changed to either use `calloc()` (which zeroes memory), or initialize the memory before using it. The value 0xCECECECE may also be seen in stack-allocated automatic variables, because in IDF most task stacks are originally allocated from the heap and in C stack memory is uninitialized by default.

If an application crashes and the exception register dump indicates that some addresses or values were 0xFEFEFEFE, this indicates it is reading heap memory after it has been freed (a “use after free bug” .) The application should be changed to not access heap memory after it has been freed.

If a call to `malloc()` or `realloc()` causes a crash because it expected to find the pattern 0xFEFEFEFE in free memory and a different pattern was found, then this indicates the app has a use-after-free bug where it is writing to memory which has already been freed.

Manual Heap Checks in Comprehensive Mode Calls to `heap_caps_check_integrity()` may print errors relating to 0xFEFEFEFE, 0xABBA1234 or 0xBAAD5678. In each case the checker is expecting to find a given pattern, and will error out if this is not found:

- For free heap blocks, the checker expects to find all bytes set to 0xFE. Any other values indicate a use-after-free bug where free memory has been incorrectly overwritten.
- For allocated heap blocks, the behaviour is the same as for *Light Impact* mode. The canary bytes 0xABBA1234 and 0xBAAD5678 are checked at the head and tail of each allocated buffer, and any variation indicates a buffer overrun/underrun.

Heap Tracing

Heap Tracing allows tracing of code which allocates/frees memory. Two tracing modes are supported:

- Standalone. In this mode trace data are kept on-board, so the size of gathered information is limited by the buffer assigned for that purposes. Analysis is done by the on-board code. There are a couple of APIs available for accessing and dumping collected info.
- Host-based. This mode does not have the limitation of the standalone mode, because trace data are sent to the host over JTAG connection using `app_trace` library. Later on they can be analysed using special tools.

Heap tracing can perform two functions:

- Leak checking: find memory which is allocated and never freed.
- Heap use analysis: show all functions that are allocating/freeing memory while the trace is running.

How To Diagnose Memory Leaks If you suspect a memory leak, the first step is to figure out which part of the program is leaking memory. Use the `xPortGetFreeHeapSize()`, `heap_caps_get_free_size()`, or *related functions* to track memory use over the life of the application. Try to narrow the leak down to a single function or sequence of functions where free memory always decreases and never recovers.

Standalone Mode Once you’ve identified the code which you think is leaking:

- In the project configuration menu, navigate to Component settings -> Heap Memory Debugging -> Heap tracing and select Standalone option (see `CONFIG_HEAP_TRACING_DEST`).
- Call the function `heap_trace_init_standalone()` early in the program, to register a buffer which can be used to record the memory trace.
- Call the function `heap_trace_start()` to begin recording all mallocs/frees in the system. Call this immediately before the piece of code which you suspect is leaking memory.
- Call the function `heap_trace_stop()` to stop the trace once the suspect piece of code has finished executing.
- Call the function `heap_trace_dump()` to dump the results of the heap trace.

An example:

```

#include "esp_heap_trace.h"

#define NUM_RECORDS 100
static heap_trace_record_t trace_record[NUM_RECORDS]; // This buffer must be in
↳internal RAM

...

void app_main()
{
    ...
    ESP_ERROR_CHECK( heap_trace_init_standalone(trace_record, NUM_RECORDS) );
    ...
}

void some_function()
{
    ESP_ERROR_CHECK( heap_trace_start(HEAP_TRACE_LEAKS) );

    do_something_you_suspect_is_leaking();

    ESP_ERROR_CHECK( heap_trace_stop() );
    heap_trace_dump();
    ...
}

```

The output from the heap trace will look something like this:

```

2 allocations trace (100 entry buffer)
32 bytes (@ 0x3ffaf214) allocated CPU 0 ccount 0x2e9b7384 caller_
↳0x400d276d:0x400d27c1
0x400d276d: leak_some_memory at /path/to/idf/examples/get-started/blink/main/./
↳blink.c:27

0x400d27c1: blink_task at /path/to/idf/examples/get-started/blink/main/./blink.c:52

8 bytes (@ 0x3ffaf804) allocated CPU 0 ccount 0x2e9b79c0 caller_
↳0x400d2776:0x400d27c1
0x400d2776: leak_some_memory at /path/to/idf/examples/get-started/blink/main/./
↳blink.c:29

0x400d27c1: blink_task at /path/to/idf/examples/get-started/blink/main/./blink.c:52

40 bytes 'leaked' in trace (2 allocations)
total allocations 2 total frees 0

```

(Above example output is using *IDF Monitor* to automatically decode PC addresses to their source files & line number.)

The first line indicates how many allocation entries are in the buffer, compared to its total size.

In `HEAP_TRACE_LEAKS` mode, for each traced memory allocation which has not already been freed a line is printed with:

- `XX bytes` is number of bytes allocated
- `@ 0x...` is the heap address returned from `malloc/calloc`.
- `CPU x` is the CPU (0 or 1) running when the allocation was made.
- `ccount 0x...` is the `CCOUNT` (CPU cycle count) register value when the allocation was made. Is different for CPU 0 vs CPU 1.
- `caller 0x...` gives the call stack of the call to `malloc()/free()`, as a list of PC addresses. These can be decoded to source files and line numbers, as shown above.

The depth of the call stack recorded for each trace entry can be configured in the project configuration menu, under Heap Memory Debugging -> Enable heap tracing -> Heap tracing stack depth. Up to 10

stack frames can be recorded for each allocation (the default is 2). Each additional stack frame increases the memory usage of each `heap_trace_record_t` record by eight bytes.

Finally, the total number of ‘leaked’ bytes (bytes allocated but not freed while trace was running) is printed, and the total number of allocations this represents.

A warning will be printed if the trace buffer was not large enough to hold all the allocations which happened. If you see this warning, consider either shortening the tracing period or increasing the number of records in the trace buffer.

Host-Based Mode Once you’ve identified the code which you think is leaking:

- In the project configuration menu, navigate to Component settings -> Heap Memory Debugging -> `CONFIG_HEAP_TRACING_DEST` and select Host-Based.
- In the project configuration menu, navigate to Component settings -> Application Level Tracing -> `CONFIG_APPTRACE_DESTINATION` and select Trace memory.
- In the project configuration menu, navigate to Component settings -> Application Level Tracing -> FreeRTOS SystemView Tracing and enable `CONFIG_SYSVIEW_ENABLE`.
- Call the function `heap_trace_init_tohost()` early in the program, to initialize JTAG heap tracing module.
- Call the function `heap_trace_start()` to begin recording all mallocs/frees in the system. Call this immediately before the piece of code which you suspect is leaking memory. In host-based mode argument to this function is ignored and heap tracing module behaves like `HEAP_TRACE_ALL` was passed: all allocations and deallocations are sent to the host.
- Call the function `heap_trace_stop()` to stop the trace once the suspect piece of code has finished executing.

An example:

```
#include "esp_heap_trace.h"

...

void app_main()
{
    ...
    ESP_ERROR_CHECK( heap_trace_init_tohost() );
    ...
}

void some_function()
{
    ESP_ERROR_CHECK( heap_trace_start(HEAP_TRACE_LEAKS) );

    do_something_you_suspect_is_leaking();

    ESP_ERROR_CHECK( heap_trace_stop() );
    ...
}
```

To gather and analyse heap trace do the following on the host:

1. Build the program and download it to the target as described in *Getting Started Guide*.
2. Run OpenOCD (see *JTAG Debugging*).

Note: In order to use this feature you need OpenOCD version `v0.10.0-esp32-20181105` or later.

3. You can use GDB to start and/or stop tracing automatically. To do this you need to prepare special `gdbinit` file:


```

target remote :3333

mon reset halt
flushregs

tb heap_trace_start
commands
mon esp32 sysview start file:///tmp/heap.svdat
c
end

tb heap_trace_stop
commands
mon esp32 sysview stop
end

c

```

Using this file GDB will connect to the target, reset it, and start tracing when program hits breakpoint at `heap_trace_start()`. Trace data will be saved to `/tmp/heap_log.svdat`. Tracing will be stopped when program hits breakpoint at `heap_trace_stop()`.

4. Run GDB using the following command `xtensa-esp32-elf-gdb -x gdbinit </path/to/program/elf>`
5. Quit GDB when program stops at `heap_trace_stop()`. Trace data are saved in `/tmp/heap.svdat`
6. Run processing script `$IDF_PATH/tools/esp_app_trace/sysviewtrace_proc.py -p -b </path/to/program/elf> /tmp/heap_log.svdat`

The output from the heap trace will look something like this:

```

Parse trace from '/tmp/heap.svdat'...
Stop parsing trace. (Timeout 0.000000 sec while reading 1 bytes!)
Process events from '['/tmp/heap.svdat']'...
[0.002244575] HEAP: Allocated 1 bytes @ 0x3ffafd8 from task "alloc" on core 0 by:
/home/user/projects/esp/esp-idf/examples/system/sysview_tracing_heap_log/main/
↪sysview_heap_log.c:47
/home/user/projects/esp/esp-idf/components/freertos/port.c:355 (discriminator 1)

[0.002258425] HEAP: Allocated 2 bytes @ 0x3ffaaffe0 from task "alloc" on core 0 by:
/home/user/projects/esp/esp-idf/examples/system/sysview_tracing_heap_log/main/
↪sysview_heap_log.c:48
/home/user/projects/esp/esp-idf/components/freertos/port.c:355 (discriminator 1)

[0.002563725] HEAP: Freed bytes @ 0x3ffaaffe0 from task "free" on core 0 by:
/home/user/projects/esp/esp-idf/examples/system/sysview_tracing_heap_log/main/
↪sysview_heap_log.c:31 (discriminator 9)
/home/user/projects/esp/esp-idf/components/freertos/port.c:355 (discriminator 1)

[0.002782950] HEAP: Freed bytes @ 0x3ffb40b8 from task "main" on core 0 by:
/home/user/projects/esp/esp-idf/components/freertos/tasks.c:4590
/home/user/projects/esp/esp-idf/components/freertos/tasks.c:4590

[0.002798700] HEAP: Freed bytes @ 0x3ffb50bc from task "main" on core 0 by:
/home/user/projects/esp/esp-idf/components/freertos/tasks.c:4590
/home/user/projects/esp/esp-idf/components/freertos/tasks.c:4590

[0.102436025] HEAP: Allocated 2 bytes @ 0x3ffaaffe0 from task "alloc" on core 0 by:
/home/user/projects/esp/esp-idf/examples/system/sysview_tracing_heap_log/main/
↪sysview_heap_log.c:47
/home/user/projects/esp/esp-idf/components/freertos/port.c:355 (discriminator 1)

[0.102449800] HEAP: Allocated 4 bytes @ 0x3ffaaffe8 from task "alloc" on core 0 by:

```

(continues on next page)

(continued from previous page)

```
/home/user/projects/esp/esp-idf/examples/system/sysview_tracing_heap_log/main/  
↪sysview_heap_log.c:48  
/home/user/projects/esp/esp-idf/components/freertos/port.c:355 (discriminator 1)  
  
[0.102666150] HEAP: Freed bytes @ 0x3ffa8e8 from task "free" on core 0 by:  
/home/user/projects/esp/esp-idf/examples/system/sysview_tracing_heap_log/main/  
↪sysview_heap_log.c:31 (discriminator 9)  
/home/user/projects/esp/esp-idf/components/freertos/port.c:355 (discriminator 1)  
  
[0.202436200] HEAP: Allocated 3 bytes @ 0x3ffa8e8 from task "alloc" on core 0 by:  
/home/user/projects/esp/esp-idf/examples/system/sysview_tracing_heap_log/main/  
↪sysview_heap_log.c:47  
/home/user/projects/esp/esp-idf/components/freertos/port.c:355 (discriminator 1)  
  
[0.202451725] HEAP: Allocated 6 bytes @ 0x3ffa9f0 from task "alloc" on core 0 by:  
/home/user/projects/esp/esp-idf/examples/system/sysview_tracing_heap_log/main/  
↪sysview_heap_log.c:48  
/home/user/projects/esp/esp-idf/components/freertos/port.c:355 (discriminator 1)  
  
[0.202667075] HEAP: Freed bytes @ 0x3ffa9f0 from task "free" on core 0 by:  
/home/user/projects/esp/esp-idf/examples/system/sysview_tracing_heap_log/main/  
↪sysview_heap_log.c:31 (discriminator 9)  
/home/user/projects/esp/esp-idf/components/freertos/port.c:355 (discriminator 1)  
  
[0.302436000] HEAP: Allocated 4 bytes @ 0x3ffa9f0 from task "alloc" on core 0 by:  
/home/user/projects/esp/esp-idf/examples/system/sysview_tracing_heap_log/main/  
↪sysview_heap_log.c:47  
/home/user/projects/esp/esp-idf/components/freertos/port.c:355 (discriminator 1)  
  
[0.302451475] HEAP: Allocated 8 bytes @ 0x3ffb40b8 from task "alloc" on core 0 by:  
/home/user/projects/esp/esp-idf/examples/system/sysview_tracing_heap_log/main/  
↪sysview_heap_log.c:48  
/home/user/projects/esp/esp-idf/components/freertos/port.c:355 (discriminator 1)  
  
[0.302667500] HEAP: Freed bytes @ 0x3ffb40b8 from task "free" on core 0 by:  
/home/user/projects/esp/esp-idf/examples/system/sysview_tracing_heap_log/main/  
↪sysview_heap_log.c:31 (discriminator 9)  
/home/user/projects/esp/esp-idf/components/freertos/port.c:355 (discriminator 1)  
  
Processing completed.  
Processed 1019 events  
===== HEAP TRACE REPORT =====  
Processed 14 heap events.  
[0.002244575] HEAP: Allocated 1 bytes @ 0x3ffa9d8 from task "alloc" on core 0 by:  
/home/user/projects/esp/esp-idf/examples/system/sysview_tracing_heap_log/main/  
↪sysview_heap_log.c:47  
/home/user/projects/esp/esp-idf/components/freertos/port.c:355 (discriminator 1)  
  
[0.102436025] HEAP: Allocated 2 bytes @ 0x3ffa8e0 from task "alloc" on core 0 by:  
/home/user/projects/esp/esp-idf/examples/system/sysview_tracing_heap_log/main/  
↪sysview_heap_log.c:47  
/home/user/projects/esp/esp-idf/components/freertos/port.c:355 (discriminator 1)  
  
[0.202436200] HEAP: Allocated 3 bytes @ 0x3ffa8e8 from task "alloc" on core 0 by:  
/home/user/projects/esp/esp-idf/examples/system/sysview_tracing_heap_log/main/  
↪sysview_heap_log.c:47  
/home/user/projects/esp/esp-idf/components/freertos/port.c:355 (discriminator 1)  
  
[0.302436000] HEAP: Allocated 4 bytes @ 0x3ffa9f0 from task "alloc" on core 0 by:  
/home/user/projects/esp/esp-idf/examples/system/sysview_tracing_heap_log/main/  
↪sysview_heap_log.c:47
```

(continues on next page)

(continued from previous page)

```
/home/user/projects/esp/esp-idf/components/freertos/port.c:355 (discriminator 1)
Found 10 leaked bytes in 4 blocks.
```

Heap Tracing To Find Heap Corruption Heap tracing can also be used to help track down heap corruption. When a region in heap is corrupted, it may be from some other part of the program which allocated memory at a nearby address.

If you have some idea at what time the corruption occurred, enabling heap tracing in `HEAP_TRACE_ALL` mode allows you to record all of the functions which allocated memory, and the addresses of the allocations.

Using heap tracing in this way is very similar to memory leak detection as described above. For memory which is allocated and not freed, the output is the same. However, records will also be shown for memory which has been freed.

Performance Impact Enabling heap tracing in menuconfig increases the code size of your program, and has a very small negative impact on performance of heap allocation/free operations even when heap tracing is not running.

When heap tracing is running, heap allocation/free operations are substantially slower than when heap tracing is stopped. Increasing the depth of stack frames recorded for each allocation (see above) will also increase this performance impact.

False-Positive Memory Leaks Not everything printed by `heap_trace_dump()` is necessarily a memory leak. Among things which may show up here, but are not memory leaks:

- Any memory which is allocated after `heap_trace_start()` but then freed after `heap_trace_stop()` will appear in the leak dump.
- Allocations may be made by other tasks in the system. Depending on the timing of these tasks, it's quite possible this memory is freed after `heap_trace_stop()` is called.
- The first time a task uses stdio - for example, when it calls `printf()` - a lock (RTOS mutex semaphore) is allocated by the libc. This allocation lasts until the task is deleted.
- Certain uses of `printf()`, such as printing floating point numbers, will allocate some memory from the heap on demand. These allocations last until the task is deleted.
- The Bluetooth, WiFi, and TCP/IP libraries will allocate heap memory buffers to handle incoming or outgoing data. These memory buffers are usually short lived, but some may be shown in the heap leak trace if the data was received/transmitted by the lower levels of the network while the leak trace was running.
- TCP connections will continue to use some memory after they are closed, because of the `TIME_WAIT` state. After the `TIME_WAIT` period has completed, this memory will be freed.

One way to differentiate between “real” and “false positive” memory leaks is to call the suspect code multiple times while tracing is running, and look for patterns (multiple matching allocations) in the heap trace output.

API Reference - Heap Tracing

Header File

- `heap/include/esp_heap_trace.h`

Functions

`esp_err_t heap_trace_init_standalone(heap_trace_record_t *record_buffer, size_t num_records)`

Initialise heap tracing in standalone mode.

This function must be called before any other heap tracing functions.

To disable heap tracing and allow the buffer to be freed, stop tracing and then call `heap_trace_init_standalone(NULL, 0)`;

Return

- ESP_ERR_NOT_SUPPORTED Project was compiled without heap tracing enabled in menuconfig.
- ESP_ERR_INVALID_STATE Heap tracing is currently in progress.
- ESP_OK Heap tracing initialised successfully.

Parameters

- `record_buffer`: Provide a buffer to use for heap trace data. Must remain valid any time heap tracing is enabled, meaning it must be allocated from internal memory not in PSRAM.
- `num_records`: Size of the heap trace buffer, as number of record structures.

`esp_err_t heap_trace_init_tohost` (void)

Initialise heap tracing in host-based mode.

This function must be called before any other heap tracing functions.

Return

- ESP_ERR_INVALID_STATE Heap tracing is currently in progress.
- ESP_OK Heap tracing initialised successfully.

`esp_err_t heap_trace_start` (*heap_trace_mode_t mode*)

Start heap tracing. All heap allocations & frees will be traced, until `heap_trace_stop()` is called.

Note `heap_trace_init_standalone()` must be called to provide a valid buffer, before this function is called.

Note Calling this function while heap tracing is running will reset the heap trace state and continue tracing.

Return

- ESP_ERR_NOT_SUPPORTED Project was compiled without heap tracing enabled in menuconfig.
- ESP_ERR_INVALID_STATE A non-zero-length buffer has not been set via `heap_trace_init_standalone()`.
- ESP_OK Tracing is started.

Parameters

- `mode`: Mode for tracing.
 - HEAP_TRACE_ALL means all heap allocations and frees are traced.
 - HEAP_TRACE_LEAKS means only suspected memory leaks are traced. (When memory is freed, the record is removed from the trace buffer.)

`esp_err_t heap_trace_stop` (void)

Stop heap tracing.

Return

- ESP_ERR_NOT_SUPPORTED Project was compiled without heap tracing enabled in menuconfig.
- ESP_ERR_INVALID_STATE Heap tracing was not in progress.
- ESP_OK Heap tracing stopped..

`esp_err_t heap_trace_resume` (void)

Resume heap tracing which was previously stopped.

Unlike `heap_trace_start()`, this function does not clear the buffer of any pre-existing trace records.

The heap trace mode is the same as when `heap_trace_start()` was last called (or HEAP_TRACE_ALL if `heap_trace_start()` was never called).

Return

- ESP_ERR_NOT_SUPPORTED Project was compiled without heap tracing enabled in menuconfig.
- ESP_ERR_INVALID_STATE Heap tracing was already started.
- ESP_OK Heap tracing resumed.

`size_t heap_trace_get_count` (void)

Return number of records in the heap trace buffer.

It is safe to call this function while heap tracing is running.

`esp_err_t heap_trace_get` (*size_t index, heap_trace_record_t *record*)

Return a raw record from the heap trace buffer.

Note It is safe to call this function while heap tracing is running, however in HEAP_TRACE_LEAK mode record indexing may skip entries unless heap tracing is stopped first.

Return

- ESP_ERR_NOT_SUPPORTED Project was compiled without heap tracing enabled in menuconfig.

- `ESP_ERR_INVALID_STATE` Heap tracing was not initialised.
- `ESP_ERR_INVALID_ARG` Index is out of bounds for current heap trace record count.
- `ESP_OK` Record returned successfully.

Parameters

- `index`: Index (zero-based) of the record to return.
- `[out] record`: Record where the heap trace record will be copied.

void **heap_trace_dump** (void)

Dump heap trace record data to stdout.

Note It is safe to call this function while heap tracing is running, however in `HEAP_TRACE_LEAK` mode the dump may skip entries unless heap tracing is stopped first.

Structures

struct heap_trace_record_t

Trace record data type. Stores information about an allocated region of memory.

Public Members

uint32_t **ccount**

CCOUNT of the CPU when the allocation was made. LSB (bit value 1) is the CPU number (0 or 1).

void ***address**

Address which was allocated.

size_t **size**

Size of the allocation.

void ***allocated_by**[`CONFIG_HEAP_TRACING_STACK_DEPTH`]

Call stack of the caller which allocated the memory.

void ***freed_by**[`CONFIG_HEAP_TRACING_STACK_DEPTH`]

Call stack of the caller which freed the memory (all zero if not freed.)

Macros

`CONFIG_HEAP_TRACING_STACK_DEPTH`

Enumerations

enum heap_trace_mode_t

Values:

`HEAP_TRACE_ALL`

`HEAP_TRACE_LEAKS`

2.7.12 High Resolution Timer

Overview

Although FreeRTOS provides software timers, these timers have a few limitations:

- Maximum resolution is equal to RTOS tick period
- Timer callbacks are dispatched from a low-priority task

Hardware timers are free from both of the limitations, but often they are less convenient to use. For example, application components may need timer events to fire at certain times in the future, but the hardware timer only contains one “compare” value used for interrupt generation. This means that some facility needs to be built on top of the hardware timer to manage the list of pending events can dispatch the callbacks for these events as corresponding hardware interrupts happen.

`esp_timer` set of APIs provide such facility. Internally, `esp_timer` uses a 32-bit hardware timer (FRC1, “legacy” timer). `esp_timer` provides one-shot and periodic timers, microsecond time resolution, and 64-bit range.

Timer callbacks are dispatched from a high-priority `esp_timer` task. Because all the callbacks are dispatched from the same task, it is recommended to only do the minimal possible amount of work from the callback itself, posting an event to a lower priority task using a queue instead.

If other tasks with priority higher than `esp_timer` are running, callback dispatching will be delayed until `esp_timer` task has a chance to run. For example, this will happen if a SPI Flash operation is in progress.

Creating and starting a timer, and dispatching the callback takes some time. Therefore there is a lower limit to the timeout value of one-shot `esp_timer`. If `esp_timer_start_once()` is called with a timeout value less than 20us, the callback will be dispatched only after approximately 20us.

Periodic `esp_timer` also imposes a 50us restriction on the minimal timer period. Periodic software timers with period of less than 50us are not practical since they would consume most of the CPU time. Consider using dedicated hardware peripherals or DMA features if you find that a timer with small period is required.

Using `esp_timer` APIs

Single timer is represented by `esp_timer_handle_t` type. Timer has a callback function associated with it. This callback function is called from the `esp_timer` task each time the timer elapses.

- To create a timer, call `esp_timer_create()`.
- To delete the timer when it is no longer needed, call `esp_timer_delete()`.

The timer can be started in one-shot mode or in periodic mode.

- To start the timer in one-shot mode, call `esp_timer_start_once()`, passing the time interval after which the callback should be called. When the callback gets called, the timer is considered to be stopped.
- To start the timer in periodic mode, call `esp_timer_start_periodic()`, passing the period with which the callback should be called. The timer keeps running until `esp_timer_stop()` is called.

Note that the timer must not be running when `esp_timer_start_once()` or `esp_timer_start_periodic()` is called. To restart a running timer, call `esp_timer_stop()` first, then call one of the start functions.

Obtaining Current Time

`esp_timer` also provides a convenience function to obtain the time passed since start-up, with microsecond precision: `esp_timer_get_time()`. This function returns the number of microseconds since `esp_timer` was initialized, which usually happens shortly before `app_main` function is called.

Unlike `gettimeofday` function, values returned by `esp_timer_get_time()`:

- Start from zero after the chip wakes up from deep sleep
- Do not have timezone or DST adjustments applied

Application Example

The following example illustrates usage of `esp_timer` APIs: [system/esp_timer](#).

API Reference

Header File

- [esp_common/include/esp_timer.h](#)

Functions

esp_err_t **esp_timer_init** (void)

Initialize esp_timer library.

Note This function is called from startup code. Applications do not need to call this function before using other esp_timer APIs.

Return

- ESP_OK on success
- ESP_ERR_NO_MEM if allocation has failed
- ESP_ERR_INVALID_STATE if already initialized
- other errors from interrupt allocator

esp_err_t **esp_timer_deinit** (void)

De-initialize esp_timer library.

Note Normally this function should not be called from applications

Return

- ESP_OK on success
- ESP_ERR_INVALID_STATE if not yet initialized

esp_err_t **esp_timer_create** (const *esp_timer_create_args_t* *create_args, *esp_timer_handle_t* *out_handle)

Create an esp_timer instance.

Note When done using the timer, delete it with esp_timer_delete function.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if some of the create_args are not valid
- ESP_ERR_INVALID_STATE if esp_timer library is not initialized yet
- ESP_ERR_NO_MEM if memory allocation fails

Parameters

- create_args: Pointer to a structure with timer creation arguments. Not saved by the library, can be allocated on the stack.
- [out] out_handle: Output, pointer to esp_timer_handle_t variable which will hold the created timer handle.

esp_err_t **esp_timer_start_once** (*esp_timer_handle_t* timer, uint64_t timeout_us)

Start one-shot timer.

Timer should not be running when this function is called.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if the handle is invalid
- ESP_ERR_INVALID_STATE if the timer is already running

Parameters

- timer: timer handle created using esp_timer_create
- timeout_us: timer timeout, in microseconds relative to the current moment

esp_err_t **esp_timer_start_periodic** (*esp_timer_handle_t* timer, uint64_t period)

Start a periodic timer.

Timer should not be running when this function is called. This function will start the timer which will trigger every 'period' microseconds.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if the handle is invalid
- ESP_ERR_INVALID_STATE if the timer is already running

Parameters

- timer: timer handle created using esp_timer_create
- period: timer period, in microseconds

esp_err_t **esp_timer_stop** (*esp_timer_handle_t* timer)

Stop the timer.

This function stops the timer previously started using `esp_timer_start_once` or `esp_timer_start_periodic`.

Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_STATE` if the timer is not running

Parameters

- `timer`: timer handle created using `esp_timer_create`

esp_err_t **esp_timer_delete** (*esp_timer_handle_t* timer)

Delete an `esp_timer` instance.

The timer must be stopped before deleting. A one-shot timer which has expired does not need to be stopped.

Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_STATE` if the timer is not running

Parameters

- `timer`: timer handle allocated using `esp_timer_create`

int64_t **esp_timer_get_time** (void)

Get time in microseconds since boot.

Return number of microseconds since `esp_timer_init` was called (this normally happens early during application startup).

int64_t **esp_timer_get_next_alarm** (void)

Get the timestamp when the next timeout is expected to occur.

Return Timestamp of the nearest timer event, in microseconds. The timebase is the same as for the values returned by `esp_timer_get_time`.

esp_err_t **esp_timer_dump** (FILE *stream)

Dump the list of timers to a stream.

If `CONFIG_ESP_TIMER_PROFILING` option is enabled, this prints the list of all the existing timers. Otherwise, only the list active timers is printed.

The format is:

```
name period alarm times_armed times_triggered total_callback_run_time
```

where:

`name` —timer name (if `CONFIG_ESP_TIMER_PROFILING` is defined), or timer pointer `period` —period of timer, in microseconds, or 0 for one-shot timer `alarm` - time of the next alarm, in microseconds since boot, or 0 if the timer is not started

The following fields are printed if `CONFIG_ESP_TIMER_PROFILING` is defined:

`times_armed` —number of times the timer was armed via `esp_timer_start_X` `times_triggered` - number of times the callback was called `total_callback_run_time` - total time taken by callback to execute, across all calls

Return

- `ESP_OK` on success
- `ESP_ERR_NO_MEM` if can not allocate temporary buffer for the output

Parameters

- `stream`: stream (such as `stdout`) to dump the information to

Structures

struct `esp_timer_create_args_t`

Timer configuration passed to `esp_timer_create`.

Public Members

`esp_timer_cb_t` callback

Function to call when timer expires.

void ***arg**

Argument to pass to the callback.

`esp_timer_dispatch_t` dispatch_method

Call the callback from task or from ISR.

const char ***name**

Timer name, used in `esp_timer_dump` function.

Type Definitions

```
typedef struct esp_timer *esp_timer_handle_t
```

Opaque type representing a single `esp_timer`.

```
typedef void (*esp_timer_cb_t)(void *arg)
```

Timer callback function type.

Parameters

- `arg`: pointer to opaque user-specific data

Enumerations

```
enum esp_timer_dispatch_t
```

Method for dispatching timer callback.

Values:

```
ESP_TIMER_TASK
```

Callback is called from timer task.

2.7.13 The himem allocation API

Overview

The ESP32 can access external SPI RAM transparently, so you can use it as normal memory in your program code. However, because the address space for external memory is limited in size, only the first 4MiB can be used as such. Access to the remaining memory is still possible, however this needs to go through a bankswitching scheme controlled by the `himem` API.

Specifically, what is implemented by the `himem` API is a bankswitching scheme. Hardware-wise, the 4MiB region for external SPI RAM is mapped into the CPU address space by a MMU, which maps a configurable 32K bank/page of external SPI RAM into each of the 32K pages in the 4MiB region accessed by the CPU. For external memories that are ≤ 4 MiB, this MMU is configured to unity mapping, effectively mapping each CPU address 1-to-1 to the external SPI RAM address.

In order to use the `himem` API, you have to enable it in the `menuconfig` using `CONFIG_SPIRAM_BANKSWITCH_ENABLE`, as well as set the amount of banks reserved for this in `CONFIG_SPIRAM_BANKSWITCH_RESERVE`. This decreases the amount of external memory allocated by functions like `malloc()`, but it allows you to use the `himem` api to map any of the remaining memory into the reserved banks.

The `himem` API is more-or-less an abstraction of the bankswitching scheme: it allows you to claim one or more banks of address space (called ‘regions’ in the API) as well as one or more of banks of memory to map into the ranges.

Example

An example doing a simple memory test of the high memory range is available in `esp-idf`: [system/himem](#)

API Reference

Header File

- [esp32/include/esp32/himem.h](#)

Functions

esp_err_t **esp_himem_alloc** (*size_t* size, *esp_himem_handle_t* *handle_out)

Allocate a block in high memory.

Return - ESP_OK if succesful

- ESP_ERR_NO_MEM if out of memory
- ESP_ERR_INVALID_SIZE if size is not a multiple of 32K

Parameters

- size: Size of the to-be-allocated block, in bytes. Note that this needs to be a multiple of the external RAM mmu block size (32K).
- [out] handle_out: Handle to be returned

esp_err_t **esp_himem_alloc_map_range** (*size_t* size, *esp_himem_rangehandle_t* *handle_out)

Allocate a memory region to map blocks into.

This allocates a contiguous CPU memory region that can be used to map blocks of physical memory into.

Return - ESP_OK if succesful

- ESP_ERR_NO_MEM if out of memory or address space
- ESP_ERR_INVALID_SIZE if size is not a multiple of 32K

Parameters

- size: Size of the range to be allocated. Note this needs to be a multiple of the external RAM mmu block size (32K).
- [out] handle_out: Handle to be returned

esp_err_t **esp_himem_map** (*esp_himem_handle_t* handle, *esp_himem_rangehandle_t* range, *size_t* ram_offset, *size_t* range_offset, *size_t* len, *int* flags, *void* **out_ptr)

Map a block of high memory into the CPUs address space.

This effectively makes the block available for read/write operations.

Note The region to be mapped needs to have offsets and sizes that are aligned to the SPI RAM MMU block size (32K)

Return - ESP_OK if the memory could be mapped

- ESP_ERR_INVALID_ARG if offset, range or len aren't MMU-block-aligned (32K)
- ESP_ERR_INVALID_SIZE if the offsets/lengths don't fit in the allocated memory or range
- ESP_ERR_INVALID_STATE if a block in the selected ram offset/length is already mapped, or if a block in the selected range offset/length already has a mapping.

Parameters

- handle: Handle to the block of memory, as given by esp_himem_alloc
- range: Range handle to map the memory in
- ram_offset: Offset into the block of physical memory of the block to map
- range_offset: Offset into the address range where the block will be mapped
- len: Length of region to map
- flags: One of ESP_HIMEM_MAPFLAG_*
- [out] out_ptr: Pointer to variable to store resulting memory pointer in

esp_err_t **esp_himem_free** (*esp_himem_handle_t* handle)

Free a block of physical memory.

This clears out the associated handle making the memory available for re-allocation again. This will only succeed if none of the memory blocks currently have a mapping.

Return - ESP_OK if the memory is succesfully freed

- ESP_ERR_INVALID_ARG if the handle still is (partially) mapped

Parameters

- handle: Handle to the block of memory, as given by esp_himem_alloc

esp_err_t **esp_himem_free_map_range** (*esp_himem_rangehandle_t* handle)

Free a mapping range.

This clears out the associated handle making the range available for re-allocation again. This will only succeed if none of the range blocks currently are used for a mapping.

Return - ESP_OK if the memory is successfully freed

- ESP_ERR_INVALID_ARG if the handle still is (partially) mapped to

Parameters

- handle: Handle to the range block, as given by `esp_himem_alloc_map_range`

esp_err_t **esp_himem_unmap** (*esp_himem_rangehandle_t* range, void *ptr, size_t len)

Unmap a region.

Return - ESP_OK if the memory is successfully unmapped,

- ESP_ERR_INVALID_ARG if ptr or len are invalid.

Parameters

- range: Range handle
- ptr: Pointer returned by `esp_himem_map`
- len: Length of the block to be unmapped. Must be aligned to the SPI RAM MMU blocksize (32K)

size_t **esp_himem_get_phys_size** (void)

Get total amount of memory under control of himem API.

Return Amount of memory, in bytes

size_t **esp_himem_get_free_size** (void)

Get free amount of memory under control of himem API.

Return Amount of free memory, in bytes

size_t **esp_himem_reserved_area_size** (void)

Get amount of SPI memory address space needed for bankswitching.

Note This is also weakly defined in `esp32/spiram.c` and returns 0 there, so if no other function in this file is used, no memory is reserved.

Return Amount of reserved area, in bytes

Macros

ESP_HIMEM_BLKSZ

ESP_HIMEM_MAPFLAG_RO

Indicates that a mapping will only be read from. Note that this is unused for now.

Type Definitions

```
typedef struct esp_himem_ramdata_t *esp_himem_handle_t
```

```
typedef struct esp_himem_rangedata_t *esp_himem_rangehandle_t
```

2.7.14 Inter-Processor Call

Overview

Due to the dual core nature of the ESP32, there are instances where a certain function must be run in the context of a particular core (e.g. allocating ISR to an interrupt source of a particular core). The IPC (Inter-Processor Call) feature allows for the execution of functions on a particular CPU.

A given function can be executed on a particular core by calling `esp_ipc_call()` or `esp_ipc_call_blocking()`. IPC is implemented via two high priority FreeRTOS tasks pinned to each CPU known as the IPC Tasks. The two IPC Tasks remain inactive (blocked) until `esp_ipc_call()` or `esp_ipc_call_blocking()` is called. When an IPC Task of a particular core is unblocked, it will preempt the current running task on that core and execute a given function.

Usage

`esp_ipc_call()` unblocks the IPC task on a particular core to execute a given function. The task that calls `esp_ipc_call()` will be blocked until the IPC Task begins execution of the given function. `esp_ipc_call_blocking()` is similar but will block the calling task until the IPC Task has completed execution of the given function.

Functions executed by IPCs must be functions of type `void func(void *arg)`. To run more complex functions which require a larger stack, the IPC tasks' stack size can be configured by modifying `CONFIG_ESP_IPC_TASK_STACK_SIZE` in `menuconfig`. The IPC API is protected by a mutex hence simultaneous IPC calls are not possible.

Care should be taken to avoid deadlock when writing functions to be executed by IPC, especially when attempting to take a mutex within the function.

API Reference

Header File

- `esp_common/include/esp_ipc.h`

Functions

`esp_err_t esp_ipc_call` (uint32_t *cpu_id*, esp_ipc_func_t *func*, void **arg*)

Execute a function on the given CPU.

Run a given function on a particular CPU. The given function must accept a `void*` argument and return `void`. The given function is run in the context of the IPC task of the CPU specified by the `cpu_id` parameter. The calling task will be blocked until the IPC task begins executing the given function. If another IPC call is ongoing, the calling task will block until the other IPC call completes. The stack size allocated for the IPC task can be configured in the “Inter-Processor Call (IPC) task stack size” setting in `menuconfig`. Increase this setting if the given function requires more stack than default.

Note In single-core mode, returns `ESP_ERR_INVALID_ARG` for `cpu_id` 1.

Return

- `ESP_ERR_INVALID_ARG` if `cpu_id` is invalid
- `ESP_ERR_INVALID_STATE` if the FreeRTOS scheduler is not running
- `ESP_OK` otherwise

Parameters

- [in] `cpu_id`: CPU where the given function should be executed (0 or 1)
- [in] `func`: Pointer to a function of type `void func(void* arg)` to be executed
- [in] `arg`: Arbitrary argument of type `void*` to be passed into the function

`esp_err_t esp_ipc_call_blocking` (uint32_t *cpu_id*, esp_ipc_func_t *func*, void **arg*)

Execute a function on the given CPU and blocks until it completes.

Run a given function on a particular CPU. The given function must accept a `void*` argument and return `void`. The given function is run in the context of the IPC task of the CPU specified by the `cpu_id` parameter. The calling task will be blocked until the IPC task completes execution of the given function. If another IPC call is ongoing, the calling task will block until the other IPC call completes. The stack size allocated for the IPC task can be configured in the “Inter-Processor Call (IPC) task stack size” setting in `menuconfig`. Increase this setting if the given function requires more stack than default.

Note In single-core mode, returns `ESP_ERR_INVALID_ARG` for `cpu_id` 1.

Return

- `ESP_ERR_INVALID_ARG` if `cpu_id` is invalid
- `ESP_ERR_INVALID_STATE` if the FreeRTOS scheduler is not running
- `ESP_OK` otherwise

Parameters

- [in] `cpu_id`: CPU where the given function should be executed (0 or 1)
- [in] `func`: Pointer to a function of type `void func(void* arg)` to be executed
- [in] `arg`: Arbitrary argument of type `void*` to be passed into the function

2.7.15 Call function with external stack

Overview

A given function can be executed with a user allocated stack space which is independent of current task stack, this mechanism can be used to save stack space wasted by tasks which call a common function with intensive stack usage such as *printf*. The given function can be called inside the shared stack space which is a callback function deferred by calling *esp_execute_shared_stack_function()*, passing that function as parameter

Usage

esp_execute_shared_stack_function() takes four arguments, a mutex object allocated by the caller, which is used to protect if the same function shares its allocated stack, a pointer to the top of stack used to that fuction, the size in bytes of stack and, a pointer to a user function where the shared stack space will reside, after calling the function, the user defined function will be deferred as a callback where functions can be called using the user allocated space without taking space from current task stack.

The usage may looks like the code below:

```
void external_stack_function(void)
{
    printf("Executing this printf from external stack! \n");
}

//Let's suppose we wanting to call printf using a separated stack space
//allowing app to reduce its stack size.
void app_main()
{
    //Allocate a stack buffer, from heap or as a static form:
    portSTACK_TYPE *shared_stack = malloc(8192 * sizeof(portSTACK_TYPE));
    assert(shared_stack != NULL);

    //Allocate a mutex to protect its usage:
    SemaphoreHandle_t printf_lock = xSemaphoreCreateMutex();
    assert(printf_lock != NULL);

    //Call the desired function using the macro helper:
    esp_execute_shared_stack_function(printf_lock,
                                     shared_stack,
                                     8192,
                                     external_stack_function);

    vSemaphoreDelete(printf_lock);
    free(shared_stack);
}
```

API Reference

Header File

- [esp_common/include/esp_expression_with_stack.h](#)

Functions

void **esp_execute_shared_stack_function** (*SemaphoreHandle_t lock*, void **stack*, size_t *stack_size*, *shared_stack_function*)

Calls user defined shared stack space function.

Note if either lock, stack or stack size is invalid, the expression will be called using the current stack.

Parameters

- `lock`: Mutex object to protect in case of shared stack
- `stack`: Pointer to user allocated stack
- `stack_size`: Size of current stack in bytes
- `function`: pointer to the shared stack function to be executed

Macros

`ESP_EXECUTE_EXPRESSION_WITH_STACK` (`lock`, `stack`, `stack_size`, `expression`)

Type Definitions

`typedef void (*shared_stack_function) (void)`

2.7.16 Interrupt allocation

Overview

The ESP32 has two cores, with 32 interrupts each. Each interrupt has a certain priority level, most (but not all) interrupts are connected to the interrupt mux. Because there are more interrupt sources than interrupts, sometimes it makes sense to share an interrupt in multiple drivers. The `esp_intr_alloc` abstraction exists to hide all these implementation details.

A driver can allocate an interrupt for a certain peripheral by calling `esp_intr_alloc` (or `esp_intr_alloc_sintrstatus`). It can use the flags passed to this function to set the type of interrupt allocated, specifying a specific level or trigger method. The interrupt allocation code will then find an applicable interrupt, use the interrupt mux to hook it up to the peripheral, and install the given interrupt handler and ISR to it.

This code has two different types of interrupts it handles differently: Shared interrupts and non-shared interrupts. The simplest of the two are non-shared interrupts: a separate interrupt is allocated per `esp_intr_alloc` call and this interrupt is solely used for the peripheral attached to it, with only one ISR that will get called. Shared interrupts can have multiple peripherals triggering it, with multiple ISRs being called when one of the peripherals attached signals an interrupt. Thus, ISRs that are intended for shared interrupts should check the interrupt status of the peripheral they service in order to see if an action is required.

Non-shared interrupts can be either level- or edge-triggered. Shared interrupts can only be level interrupts (because of the chance of missed interrupts when edge interrupts are used.) (The logic behind this: DevA and DevB share an int. DevB signals an int. Int line goes high. ISR handler calls code for DevA -> does nothing. ISR handler calls code for DevB, but while doing that, DevA signals an int. ISR DevB is done, clears int for DevB, exits interrupt code. Now an interrupt for DevA is still pending, but because the int line never went low (DevA kept it high even when the int for DevB was cleared) the interrupt is never serviced.)

Multicore issues

Peripherals that can generate interrupts can be divided in two types:

- External peripherals, within the ESP32 but outside the Xtensa cores themselves. Most ESP32 peripherals are of this type.
- Internal peripherals, part of the Xtensa CPU cores themselves.

Interrupt handling differs slightly between these two types of peripherals.

Internal peripheral interrupts Each Xtensa CPU core has its own set of six internal peripherals:

- Three timer comparators
- A performance monitor
- Two software interrupts.

Internal interrupt sources are defined in `esp_intr_alloc.h` as `ETS_INTERNAL_*_INTR_SOURCE`.

These peripherals can only be configured from the core they are associated with. When generating an interrupt, the interrupt they generate is hard-wired to their associated core; it's not possible to have e.g. an internal timer comparator of one core generate an interrupt on another core. That is why these sources can only be managed using a task running on that specific core. Internal interrupt sources are still allocatable using `esp_intr_alloc` as normal, but they cannot be shared and will always have a fixed interrupt level (namely, the one associated in hardware with the peripheral).

External Peripheral Interrupts The remaining interrupt sources are from external peripherals. These are defined in `soc/soc.h` as `ETS_*_INTR_SOURCE`.

Non-internal interrupt slots in both CPU cores are wired to an interrupt multiplexer, which can be used to route any external interrupt source to any of these interrupt slots.

- Allocating an external interrupt will always allocate it on the core that does the allocation.
- Freeing an external interrupt must always happen on the same core it was allocated on.
- Disabling and enabling external interrupts from another core is allowed.
- Multiple external interrupt sources can share an interrupt slot by passing `ESP_INTR_FLAG_SHARED` as a flag to `esp_intr_alloc()`.

Care should be taken when calling `esp_intr_alloc()` from a task which is not pinned to a core. During task switching, these tasks can migrate between cores. Therefore it is impossible to tell which CPU the interrupt is allocated on, which makes it difficult to free the interrupt handle and may also cause debugging difficulties. It is advised to use `xTaskCreatePinnedToCore()` with a specific `CoreID` argument to create tasks that will allocate interrupts. In the case of internal interrupt sources, this is required.

IRAM-Safe Interrupt Handlers

The `ESP_INTR_FLAG_IRAM` flag registers an interrupt handler that always runs from IRAM (and reads all its data from DRAM), and therefore does not need to be disabled during flash erase and write operations.

This is useful for interrupts which need a guaranteed minimum execution latency, as flash write and erase operations can be slow (erases can take tens or hundreds of milliseconds to complete).

It can also be useful to keep an interrupt handler in IRAM if it is called very frequently, to avoid flash cache misses.

Refer to the [SPI flash API documentation](#) for more details.

Multiple Handlers Sharing A Source

Several handlers can be assigned to a same source, given that all handlers are allocated using the `ESP_INTR_FLAG_SHARED` flag. They'll be all allocated to the interrupt, which the source is attached to, and called sequentially when the source is active. The handlers can be disabled and freed individually. The source is attached to the interrupt (enabled), if one or more handlers are enabled, otherwise detached. A handler will never be called when disabled, while **its source may still be triggered** if any one of its handler enabled.

Sources attached to non-shared interrupt do not support this feature.

Though the framework support this feature, you have to use it *very carefully*. There usually exist 2 ways to stop a interrupt from being triggered: *disable the source* or *mask peripheral interrupt status*. IDF only handles the enabling and disabling of the source itself, leaving status and mask bits to be handled by users. **Status bits should always be masked before the handler responsible for it is disabled, or the status should be handled in other enabled interrupt properly**. You may leave some status bits unhandled if you just disable one of all the handlers without masking the status bits, which causes the interrupt to trigger infinitely resulting in a system crash.

API Reference

Header File

- [esp32/include/esp_intr_alloc.h](#)

Functions

esp_err_t **esp_intr_mark_shared** (int *intno*, int *cpu*, bool *is_in_iram*)

Mark an interrupt as a shared interrupt.

This will mark a certain interrupt on the specified CPU as an interrupt that can be used to hook shared interrupt handlers to.

Return ESP_ERR_INVALID_ARG if *cpu* or *intno* is invalid ESP_OK otherwise

Parameters

- *intno*: The number of the interrupt (0-31)
- *cpu*: CPU on which the interrupt should be marked as shared (0 or 1)
- *is_in_iram*: Shared interrupt is for handlers that reside in IRAM and the int can be left enabled while the flash cache is disabled.

esp_err_t **esp_intr_reserve** (int *intno*, int *cpu*)

Reserve an interrupt to be used outside of this framework.

This will mark a certain interrupt on the specified CPU as reserved, not to be allocated for any reason.

Return ESP_ERR_INVALID_ARG if *cpu* or *intno* is invalid ESP_OK otherwise

Parameters

- *intno*: The number of the interrupt (0-31)
- *cpu*: CPU on which the interrupt should be marked as shared (0 or 1)

esp_err_t **esp_intr_alloc** (int *source*, int *flags*, *intr_handler_t* *handler*, void **arg*, *intr_handle_t* **ret_handle*)

Allocate an interrupt with the given parameters.

This finds an interrupt that matches the restrictions as given in the flags parameter, maps the given interrupt source to it and hooks up the given interrupt handler (with optional argument) as well. If needed, it can return a handle for the interrupt as well.

The interrupt will always be allocated on the core that runs this function.

If ESP_INTR_FLAG_IRAM flag is used, and handler address is not in IRAM or RTC_FAST_MEM, then ESP_ERR_INVALID_ARG is returned.

Return ESP_ERR_INVALID_ARG if the combination of arguments is invalid. ESP_ERR_NOT_FOUND

No free interrupt found with the specified flags ESP_OK otherwise

Parameters

- *source*: The interrupt source. One of the ETS_*_INTR_SOURCE interrupt mux sources, as defined in soc/soc.h, or one of the internal ETS_INTERNAL_*_INTR_SOURCE sources as defined in this header.
- *flags*: An ORred mask of the ESP_INTR_FLAG_* defines. These restrict the choice of interrupts that this routine can choose from. If this value is 0, it will default to allocating a non-shared interrupt of level 1, 2 or 3. If this is ESP_INTR_FLAG_SHARED, it will allocate a shared interrupt of level 1. Setting ESP_INTR_FLAG_INTRDISABLED will return from this function with the interrupt disabled.
- *handler*: The interrupt handler. Must be NULL when an interrupt of level >3 is requested, because these types of interrupts aren't C-callable.
- *arg*: Optional argument for passed to the interrupt handler
- *ret_handle*: Pointer to an *intr_handle_t* to store a handle that can later be used to request details or free the interrupt. Can be NULL if no handle is required.

esp_err_t **esp_intr_alloc_intrstatus** (int *source*, int *flags*, uint32_t *intrstatusreg*, uint32_t *intrstatusmask*, *intr_handler_t* *handler*, void **arg*, *intr_handle_t* **ret_handle*)

Allocate an interrupt with the given parameters.

This essentially does the same as *esp_intr_alloc*, but allows specifying a register and mask combo. For shared interrupts, the handler is only called if a read from the specified register, ANDed with the mask, returns non-zero. By passing an interrupt status register address and a fitting mask, this can be used to accelerate interrupt

handling in the case a shared interrupt is triggered; by checking the interrupt statuses first, the code can decide which ISRs can be skipped

Return ESP_ERR_INVALID_ARG if the combination of arguments is invalid. ESP_ERR_NOT_FOUND
No free interrupt found with the specified flags ESP_OK otherwise

Parameters

- **source**: The interrupt source. One of the ETS_*_INTR_SOURCE interrupt mux sources, as defined in soc/soc.h, or one of the internal ETS_INTERNAL_*_INTR_SOURCE sources as defined in this header.
- **flags**: An ORred mask of the ESP_INTR_FLAG_* defines. These restrict the choice of interrupts that this routine can choose from. If this value is 0, it will default to allocating a non-shared interrupt of level 1, 2 or 3. If this is ESP_INTR_FLAG_SHARED, it will allocate a shared interrupt of level 1. Setting ESP_INTR_FLAG_INTRDISABLED will return from this function with the interrupt disabled.
- **intrstatusreg**: The address of an interrupt status register
- **intrstatusmask**: A mask. If a read of address intrstatusreg has any of the bits that are 1 in the mask set, the ISR will be called. If not, it will be skipped.
- **handler**: The interrupt handler. Must be NULL when an interrupt of level >3 is requested, because these types of interrupts aren't C-callable.
- **arg**: Optional argument for passed to the interrupt handler
- **ret_handle**: Pointer to an intr_handle_t to store a handle that can later be used to request details or free the interrupt. Can be NULL if no handle is required.

esp_err_t esp_intr_free (intr_handle_t handle)

Disable and free an interrupt.

Use an interrupt handle to disable the interrupt and release the resources associated with it. If the current core is not the core that registered this interrupt, this routine will be assigned to the core that allocated this interrupt, blocking and waiting until the resource is successfully released.

Note When the handler shares its source with other handlers, the interrupt status bits it's responsible for should be managed properly before freeing it. see esp_intr_disable for more details. Please do not call this function in esp_ipc_call_blocking.

Return ESP_ERR_INVALID_ARG the handle is NULL ESP_FAIL failed to release this handle ESP_OK otherwise

Parameters

- **handle**: The handle, as obtained by esp_intr_alloc or esp_intr_alloc_intrstatus

int esp_intr_get_cpu (intr_handle_t handle)

Get CPU number an interrupt is tied to.

Return The core number where the interrupt is allocated

Parameters

- **handle**: The handle, as obtained by esp_intr_alloc or esp_intr_alloc_intrstatus

int esp_intr_get_intno (intr_handle_t handle)

Get the allocated interrupt for a certain handle.

Return The interrupt number

Parameters

- **handle**: The handle, as obtained by esp_intr_alloc or esp_intr_alloc_intrstatus

esp_err_t esp_intr_disable (intr_handle_t handle)

Disable the interrupt associated with the handle.

Note

1. For local interrupts (ESP_INTERNAL_* sources), this function has to be called on the CPU the interrupt is allocated on. Other interrupts have no such restriction.
2. When several handlers sharing a same interrupt source, interrupt status bits, which are handled in the handler to be disabled, should be masked before the disabling, or handled in other enabled interrupts properly. Miss of interrupt status handling will cause infinite interrupt calls and finally system crash.

Return ESP_ERR_INVALID_ARG if the combination of arguments is invalid. ESP_OK otherwise

Parameters

- `handle`: The handle, as obtained by `esp_intr_alloc` or `esp_intr_alloc_intrstatus`

esp_err_t **esp_intr_enable** (*intr_handle_t* handle)

Enable the interrupt associated with the handle.

Note For local interrupts (`ESP_INTERNAL_*` sources), this function has to be called on the CPU the interrupt is allocated on. Other interrupts have no such restriction.

Return `ESP_ERR_INVALID_ARG` if the combination of arguments is invalid. `ESP_OK` otherwise

Parameters

- `handle`: The handle, as obtained by `esp_intr_alloc` or `esp_intr_alloc_intrstatus`

esp_err_t **esp_intr_set_in_iram** (*intr_handle_t* handle, bool *is_in_iram*)

Set the “in IRAM” status of the handler.

Note Does not work on shared interrupts.

Return `ESP_ERR_INVALID_ARG` if the combination of arguments is invalid. `ESP_OK` otherwise

Parameters

- `handle`: The handle, as obtained by `esp_intr_alloc` or `esp_intr_alloc_intrstatus`
- `is_in_iram`: Whether the handler associated with this handle resides in IRAM. Handlers residing in IRAM can be called when cache is disabled.

void **esp_intr_noniram_disable** (void)

Disable interrupts that aren't specifically marked as running from IRAM.

void **esp_intr_noniram_enable** (void)

Re-enable interrupts disabled by `esp_intr_noniram_disable`.

Macros

ESP_INTR_FLAG_LEVEL1

Interrupt allocation flags.

These flags can be used to specify which interrupt qualities the code calling `esp_intr_alloc*` needs. Accept a Level 1 interrupt vector (lowest priority)

ESP_INTR_FLAG_LEVEL2

Accept a Level 2 interrupt vector.

ESP_INTR_FLAG_LEVEL3

Accept a Level 3 interrupt vector.

ESP_INTR_FLAG_LEVEL4

Accept a Level 4 interrupt vector.

ESP_INTR_FLAG_LEVEL5

Accept a Level 5 interrupt vector.

ESP_INTR_FLAG_LEVEL6

Accept a Level 6 interrupt vector.

ESP_INTR_FLAG_NMI

Accept a Level 7 interrupt vector (highest priority)

ESP_INTR_FLAG_SHARED

Interrupt can be shared between ISRs.

ESP_INTR_FLAG_EDGE

Edge-triggered interrupt.

ESP_INTR_FLAG_IRAM

ISR can be called if cache is disabled.

ESP_INTR_FLAG_INTRDISABLED

Return with this interrupt disabled.

ESP_INTR_FLAG_LOWMED

Low and medium prio interrupts. These can be handled in C.

ESP_INTR_FLAG_HIGH

High level interrupts. Need to be handled in assembly.

ESP_INTR_FLAG_LEVELMASK

Mask for all level flags.

ETS_INTERNAL_TIMER0_INTR_SOURCE

Xtensa timer 0 interrupt source.

The `esp_intr_alloc*` functions can allocate an int for all `ETS_*_INTR_SOURCE` interrupt sources that are routed through the interrupt mux. Apart from these sources, each core also has some internal sources that do not pass through the interrupt mux. To allocate an interrupt for these sources, pass these pseudo-sources to the functions.

ETS_INTERNAL_TIMER1_INTR_SOURCE

Xtensa timer 1 interrupt source.

ETS_INTERNAL_TIMER2_INTR_SOURCE

Xtensa timer 2 interrupt source.

ETS_INTERNAL_SW0_INTR_SOURCE

Software int source 1.

ETS_INTERNAL_SW1_INTR_SOURCE

Software int source 2.

ETS_INTERNAL_PROFILING_INTR_SOURCE

Int source for profiling.

ETS_INTERNAL_INTR_SOURCE_OFF

ESP_INTR_ENABLE (inum)

ESP_INTR_DISABLE (inum)

Type Definitions

```
typedef void (*intr_handler_t) (void *arg)
```

```
typedef struct intr_handle_data_t intr_handle_data_t
```

```
typedef intr_handle_data_t *intr_handle_t
```

2.7.17 Logging library

Overview

The logging library provides two ways for setting log verbosity:

- **At compile time:** in menuconfig, set the verbosity level using the option `CONFIG_LOG_DEFAULT_LEVEL`. All logging statements for verbosity levels higher than `CONFIG_LOG_DEFAULT_LEVEL` will be removed by the preprocessor.
- **At runtime:** all logs for verbosity levels lower than `CONFIG_LOG_DEFAULT_LEVEL` are enabled by default. The function `esp_log_level_set()` can be used to set a logging level on a per module basis. Modules are identified by their tags, which are human-readable ASCII zero-terminated strings.

There are the following verbosity levels:

- Error (lowest)
- Warning
- Info
- Debug
- Verbose (highest)

Note: The function `esp_log_level_set()` cannot set logging levels higher than specified by `CONFIG_LOG_DEFAULT_LEVEL`. To increase log level for a specific file at compile time, use the macro `LOG_LOCAL_LEVEL` (see the details below).

How to use this library

In each C file that uses logging functionality, define the TAG variable as shown below:

```
static const char* TAG = "MyModule";
```

Then use one of logging macros to produce output, e.g:

```
ESP_LOGW(TAG, "Baud rate error %.1f%%. Requested: %d baud, actual: %d baud", error_
↪* 100, baud_req, baud_real);
```

Several macros are available for different verbosity levels:

- `ESP_LOGE` - error (lowest)
- `ESP_LOGW` - warning
- `ESP_LOGI` - info
- `ESP_LOGD` - debug
- `ESP_LOGV` - verbose (highest)

Additionally, there are `ESP_EARLY_LOGx` versions for each of these macros, e.g., `ESP_EARLY_LOGE`. These versions have to be used explicitly in the early startup code only, before heap allocator and syscalls have been initialized. Normal `ESP_LOGx` macros can also be used while compiling the bootloader, but they will fall back to the same implementation as `ESP_EARLY_LOGx` macros.

To override default verbosity level at file or component scope, define the `LOG_LOCAL_LEVEL` macro.

At file scope, define it before including `esp_log.h`, e.g.:

```
#define LOG_LOCAL_LEVEL ESP_LOG_VERBOSE
#include "esp_log.h"
```

At component scope, define it in the component makefile:

```
CFLAGS += -D LOG_LOCAL_LEVEL=ESP_LOG_DEBUG
```

To configure logging output per module at runtime, add calls to the function `esp_log_level_set()` as follows:

```
esp_log_level_set("", ESP_LOG_ERROR);           // set all components to ERROR level
esp_log_level_set("wifi", ESP_LOG_WARN);       // enable WARN logs from WiFi stack
esp_log_level_set("dhcpc", ESP_LOG_INFO);      // enable INFO logs from DHCP client
```

Logging to Host via JTAG By default, the logging library uses the `vprintf`-like function to write formatted output to the dedicated UART. By calling a simple API, all log output may be routed to JTAG instead, making logging several times faster. For details, please refer to Section [Logging to Host](#).

Application Example

The logging library is commonly used by most esp-idf components and examples. For demonstration of log functionality, check ESP-IDF's [examples](#) directory. The most relevant examples that deal with logging are the following:

- `system/ota`
- `storage/sd_card`
- `protocols/https_request`

API Reference

Header File

- [log/include/esp_log.h](#)

Functions

void **esp_log_level_set** (**const** char **tag*, *esp_log_level_t level*)
Set log level for given tag.

If logging for given component has already been enabled, changes previous setting.

Note that this function can not raise log level above the level set using CONFIG_LOG_DEFAULT_LEVEL setting in menuconfig.

To raise log level above the default one for a given file, define LOG_LOCAL_LEVEL to one of the ESP_LOG_* values, before including esp_log.h in this file.

Parameters

- *tag*: Tag of the log entries to enable. Must be a non-NULL zero terminated string. Value "*" resets log level for all tags to the given value.
- *level*: Selects log level to enable. Only logs at this and lower verbosity levels will be shown.

vprintf_like_t **esp_log_set_vprintf** (*vprintf_like_t func*)
Set function used to output log entries.

By default, log output goes to UART0. This function can be used to redirect log output to some other destination, such as file or network. Returns the original log handler, which may be necessary to return output to the previous destination.

Return *func* old Function used for output.

Parameters

- *func*: new Function used for output. Must have same signature as vprintf.

uint32_t **esp_log_timestamp** (void)
Function which returns timestamp to be used in log output.

This function is used in expansion of ESP_LOGx macros. In the 2nd stage bootloader, and at early application startup stage this function uses CPU cycle counter as time source. Later when FreeRTOS scheduler start running, it switches to FreeRTOS tick count.

For now, we ignore millisecond counter overflow.

Return timestamp, in milliseconds

char ***esp_log_system_timestamp** (void)
Function which returns system timestamp to be used in log output.

This function is used in expansion of ESP_LOGx macros to print the system time as "HH:MM:SS.sss". The system time is initialized to 0 on startup, this can be set to the correct time with an SNTP sync, or manually with standard POSIX time functions.

Currently this will not get used in logging from binary blobs (i.e WiFi & Bluetooth libraries), these will still print the RTOS tick time.

Return timestamp, in "HH:MM:SS.sss"

uint32_t **esp_log_early_timestamp** (void)
Function which returns timestamp to be used in log output.

This function uses HW cycle counter and does not depend on OS, so it can be safely used after application crash.

Return timestamp, in milliseconds

void **esp_log_write** (*esp_log_level_t level*, **const** char **tag*, **const** char **format*, ...)
Write message into the log.

This function is not intended to be used directly. Instead, use one of `ESP_LOGE`, `ESP_LOGW`, `ESP_LOGI`, `ESP_LOGD`, `ESP_LOGV` macros.

This function or these macros should not be used from an interrupt.

void `esp_log_writev` (*esp_log_level_t* level, const char *tag, const char *format, va_list args)
Write message into the log, va_list variant.

This function is provided to ease integration toward other logging framework, so that `esp_log` can be used as a log sink.

See `esp_log_write()`

Macros

ESP_LOG_BUFFER_HEX_LEVEL (tag, buffer, buff_len, level)

Log a buffer of hex bytes at specified level, separated into 16 bytes each line.

Parameters

- tag: description tag
- buffer: Pointer to the buffer array
- buff_len: length of buffer in bytes
- level: level of the log

ESP_LOG_BUFFER_CHAR_LEVEL (tag, buffer, buff_len, level)

Log a buffer of characters at specified level, separated into 16 bytes each line. Buffer should contain only printable characters.

Parameters

- tag: description tag
- buffer: Pointer to the buffer array
- buff_len: length of buffer in bytes
- level: level of the log

ESP_LOG_BUFFER_HEXDUMP (tag, buffer, buff_len, level)

Dump a buffer to the log at specified level.

The dump log shows just like the one below:

```

W (195) log_example: 0x3ffb4280  45 53 50 33 32 20 69 73  20 67 72 65 61 74_
↪2c 20 |ESP32 is great, |
W (195) log_example: 0x3ffb4290  77 6f 72 6b 69 6e 67 20  61 6c 6f 6e 67 20_
↪77 69 |working along wi|
W (205) log_example: 0x3ffb42a0  74 68 20 74 68 65 20 49  44 46 2e 00      _
↪      |th the IDF..|

```

It is highly recommend to use terminals with over 102 text width.

Parameters

- tag: description tag
- buffer: Pointer to the buffer array
- buff_len: length of buffer in bytes
- level: level of the log

ESP_LOG_BUFFER_HEX (tag, buffer, buff_len)

Log a buffer of hex bytes at Info level.

See `esp_log_buffer_hex_level`

Parameters

- tag: description tag
- buffer: Pointer to the buffer array
- buff_len: length of buffer in bytes

ESP_LOG_BUFFER_CHAR (tag, buffer, buff_len)

Log a buffer of characters at Info level. Buffer should contain only printable characters.

See `esp_log_buffer_char_level`

Parameters

- `tag`: description tag
- `buffer`: Pointer to the buffer array
- `buff_len`: length of buffer in bytes

ESP_EARLY_LOGE (`tag, format, ...`)macro to output logs in startup code, before heap allocator and syscalls have been initialized. log at `ESP_LOG_ERROR` level.**See** `printf,ESP_LOGE`**ESP_EARLY_LOGW** (`tag, format, ...`)macro to output logs in startup code at `ESP_LOG_WARN` level.**See** `ESP_EARLY_LOGE,ESP_LOGE,printf`**ESP_EARLY_LOGI** (`tag, format, ...`)macro to output logs in startup code at `ESP_LOG_INFO` level.**See** `ESP_EARLY_LOGE,ESP_LOGE,printf`**ESP_EARLY_LOGD** (`tag, format, ...`)macro to output logs in startup code at `ESP_LOG_DEBUG` level.**See** `ESP_EARLY_LOGE,ESP_LOGE,printf`**ESP_EARLY_LOGV** (`tag, format, ...`)macro to output logs in startup code at `ESP_LOG_VERBOSE` level.**See** `ESP_EARLY_LOGE,ESP_LOGE,printf`**ESP_LOG_EARLY_IMPL** (`tag, format, log_level, log_tag_letter, ...`)**ESP_LOGE** (`tag, format, ...`)**ESP_LOGW** (`tag, format, ...`)**ESP_LOGI** (`tag, format, ...`)**ESP_LOGD** (`tag, format, ...`)**ESP_LOGV** (`tag, format, ...`)**ESP_LOG_LEVEL_LOCAL** (`level, tag, format, ...`)

runtime macro to output logs at a specified level.

See `printf` runtime macro to output logs at a specified level. Also check the level with `LOG_LOCAL_LEVEL`.**See** `printf,ESP_LOG_LEVEL`**Parameters**

- `tag`: tag of the log, which can be used to change the log level by `esp_log_level_set` at runtime.
- `level`: level of the output log.
- `format`: format of the output log. see `printf`
- `...`: variables to be replaced into the log. see `printf`

Type Definitions**typedef** `int (*vprintf_like_t) (const char *, va_list)`**Enumerations****enum** `esp_log_level_t`

Log level.

*Values:***ESP_LOG_NONE**

No log output

ESP_LOG_ERROR

Critical errors, software module can not recover on its own

ESP_LOG_WARN

Error conditions from which recovery measures have been taken

ESP_LOG_INFO

Information messages which describe normal flow of events

ESP_LOG_DEBUG

Extra information which is not necessary for normal use (values, pointers, sizes, etc).

ESP_LOG_VERBOSE

Bigger chunks of debugging information, or frequent messages which can potentially flood the output.

2.7.18 Miscellaneous System APIs

Software reset

To perform software reset of the chip, `esp_restart()` function is provided. When the function is called, execution of the program will stop, both CPUs will be reset, application will be loaded by the bootloader and started again.

Additionally, `esp_register_shutdown_handler()` function is provided to register a routine which needs to be called prior to restart (when done by `esp_restart()`). This is similar to the functionality of `atexit` POSIX function.

Reset reason

ESP-IDF application can be started or restarted due to a variety of reasons. To get the last reset reason, call `esp_reset_reason()` function. See description of `esp_reset_reason_t` for the list of possible reset reasons.

Heap memory

Two heap memory related functions are provided:

- `esp_get_free_heap_size()` returns the current size of free heap memory
- `esp_get_minimum_free_heap_size()` returns the minimum size of free heap memory that was available during program execution.

Note that ESP-IDF supports multiple heaps with different capabilities. Functions mentioned in this section return the size of heap memory which can be allocated using `malloc` family of functions. For further information about heap memory see [Heap Memory Allocation](#).

Random number generation

ESP32 contains a hardware random number generator, values from it can be obtained using `esp_random()`.

When Wi-Fi or Bluetooth are enabled, numbers returned by hardware random number generator (RNG) can be considered true random numbers. Without Wi-Fi or Bluetooth enabled, hardware RNG is a pseudo-random number generator. At startup, ESP-IDF bootloader seeds the hardware RNG with entropy, but care must be taken when reading random values between the start of `app_main` and initialization of Wi-Fi or Bluetooth drivers.

MAC Address

These APIs allow querying and customizing MAC addresses used by Wi-Fi, Bluetooth, and Ethernet drivers.

ESP32 has up to 4 network interfaces: Wi-Fi station, Wi-Fi AP, Ethernet, and Bluetooth. Each of these interfaces needs to have a MAC address assigned to it. In ESP-IDF these addresses are calculated from *Base MAC address*. Base MAC address can be initialized with factory-programmed value from EFUSE, or with a user-defined value. In addition to setting the base MAC address, applications can specify the way in which MAC addresses are allocated to devices. See [Number of universally administered MAC address](#) section for more details.

| Interface | MAC address (4 universally administered) | MAC address (2 universally administered) |
|---------------|--|--|
| Wi-Fi Station | base_mac | base_mac |
| Wi-Fi SoftAP | base_mac, +1 to the last octet | base_mac, first octet randomized |
| Bluetooth | base_mac, +2 to the last octet | base_mac, +1 to the last octet |
| Ethernet | base_mac, +3 to the last octet | base_mac, +1 to the last octet, first octet randomized |

Base MAC address Wi-Fi, Bluetooth, and Ethernet drivers use `esp_read_mac()` function to get MAC address for a specific interface.

By default, this function will use MAC address factory programmed in BLK0 of EFUSE as the base MAC address. MAC addresses of each interface will be calculated according to the table above.

Applications which don't use MAC address factory programmed into BLK0 of EFUSE can modify base MAC address used by `esp_read_mac()` using a call to `esp_base_mac_addr_set()`. Custom value of MAC address can come from application defined storage, such as Flash, NVS, etc. Note that the call to `esp_base_mac_addr_set()` needs to happen before network protocol stacks are initialized, for example, early in `app_main`.

Custom MAC address in BLK3 of EFUSE To facilitate usage of custom MAC addresses, ESP-IDF provides `esp_efuse_mac_get_custom()` function, which loads MAC address from BLK3 of EFUSE. This function assumes that custom MAC address is stored in BLK3 of EFUSE (EFUSE_BLK3_RDATA0, EFUSE_BLK3_RDATA1, EFUSE_BLK3_RDATA2, EFUSE_BLK3_RDATA3, EFUSE_BLK3_RDATA4, EFUSE_BLK3_RDATA5 registers) in the following format:

| Field | # of bits | Range of bits | Notes |
|-----------------|-----------|---------------|------------------------------|
| Version | 8 | 191:184 | 0: invalid, others — valid |
| Reserved | 128 | 183:56 | |
| MAC address | 48 | 55:8 | |
| MAC address CRC | 8 | 7:0 | CRC-8-CCITT, polynomial 0x07 |

Once MAC address has been obtained using `esp_efuse_mac_get_custom()`, call `esp_base_mac_addr_set()` to set this MAC address as base MAC address.

Number of universally administered MAC address Several MAC addresses (universally administered by IEEE) are uniquely assigned to the networking interfaces (Wi-Fi/BT/Ethernet). The final octet of each universally administered MAC address increases by one. Only the first one of them (which is called base MAC address) is stored in EFUSE or external storage, the others are generated from it. Here, 'generate' means adding 0, 1, 2 and 3 (respectively) to the final octet of the base MAC address.

If the universally administered MAC addresses are not enough for all of the networking interfaces, locally administered MAC addresses which are derived from universally administered MAC addresses are assigned to the rest of networking interfaces.

See [this article](#) for the definition of local and universally administered MAC addresses.

The number of universally administered MAC address can be configured using `CONFIG_ESP32_UNIVERSAL_MAC_ADDRESSES`.

If the number of universal MAC addresses is two, only two interfaces (Wi-Fi Station and Bluetooth) receive a universally administered MAC address. These are generated sequentially by adding 0 and 1 (respectively) to the base MAC address. The remaining two interfaces (Wi-Fi SoftAP and Ethernet) receive local MAC addresses. These are derived from the universal Wi-Fi station and Bluetooth MAC addresses, respectively.

If the number of universal MAC addresses is four, all four interfaces (Wi-Fi Station, Wi-Fi SoftAP, Bluetooth and Ethernet) receive a universally administered MAC address. These are generated sequentially by adding 0, 1, 2 and 3 (respectively) to the final octet of the base MAC address.

When using the default (Espressif-assigned) base MAC address, either setting can be used. When using a custom universal MAC address range, the correct setting will depend on the allocation of MAC addresses in this range (either 2 or 4 per device.)

Chip version

`esp_chip_info()` function fills `esp_chip_info_t` structure with information about the chip. This includes the chip revision, number of CPU cores, and a bit mask of features enabled in the chip.

SDK version

`esp_get_idf_version()` returns a string describing the IDF version which was used to compile the application. This is the same value as the one available through `IDF_VER` variable of the build system. The version string generally has the format of `git describe output`.

To get the version at build time, additional version macros are provided. They can be used to enable or disable parts of the program depending on IDF version.

- `ESP_IDF_VERSION_MAJOR`, `ESP_IDF_VERSION_MINOR`, `ESP_IDF_VERSION_PATCH` are defined to integers representing major, minor, and patch version.
- `ESP_IDF_VERSION_VAL` and `ESP_IDF_VERSION` can be used when implementing version checks:

```
#include "esp_idf_version.h"

#if ESP_IDF_VERSION >= ESP_IDF_VERSION_VAL(4, 0, 0)
    // enable functionality present in IDF v4.0
#endif
```

App version

Application version is stored in `esp_app_desc_t` structure. It is located in DROM sector and has a fixed offset from the beginning of the binary file. The structure is located after `esp_image_header_t` and `esp_image_segment_header_t` structures. The field `version` has string type and max length 32 chars.

To set version in your project manually you need to set `PROJECT_VER` variable in your project `CMakeLists.txt/Makefile`:

- In application `CMakeLists.txt` put `set(PROJECT_VER "0.1.0.1")` before including `project.cmake`.

(For legacy GNU Make build system: in application `Makefile` put `PROJECT_VER = "0.1.0.1"` before including `project.mk`.)

If `PROJECT_VER` variable is not set in the project then it will be retrieved from either `$(PROJECT_PATH)/version.txt` file (if present) else using `git describe`. If neither is available then `PROJECT_VER` will be set to "1". Application can make use of this by calling `esp_ota_get_app_description()` or `esp_ota_get_partition_description()` functions.

API Reference

Header File

- [esp_common/include/esp_system.h](#)

Functions

esp_err_t **esp_register_shutdown_handler** (*shutdown_handler_t* handle)

Register shutdown handler.

This function allows you to register a handler that gets invoked before the application is restarted using `esp_restart` function.

Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_STATE` if the handler has already been registered
- `ESP_ERR_NO_MEM` if no more shutdown handler slots are available

Parameters

- `handle`: function to execute on restart

esp_err_t **esp_unregister_shutdown_handler** (*shutdown_handler_t* handle)

Unregister shutdown handler.

This function allows you to unregister a handler which was previously registered using `esp_register_shutdown_handler` function.

- `ESP_OK` on success
- `ESP_ERR_INVALID_STATE` if the given handler hasn't been registered before

void **esp_restart** (void)

Restart PRO and APP CPUs.

This function can be called both from PRO and APP CPUs. After successful restart, CPU reset reason will be `SW_CPU_RESET`. Peripherals (except for WiFi, BT, UART0, SPI1, and legacy timers) are not reset. This function does not return.

esp_reset_reason_t **esp_reset_reason** (void)

Get reason of last reset.

Return See description of `esp_reset_reason_t` for explanation of each value.

uint32_t **esp_get_free_heap_size** (void)

Get the size of available heap.

Note that the returned value may be larger than the maximum contiguous block which can be allocated.

Return Available heap size, in bytes.

uint32_t **esp_get_free_internal_heap_size** (void)

Get the size of available internal heap.

Note that the returned value may be larger than the maximum contiguous block which can be allocated.

Return Available internal heap size, in bytes.

uint32_t **esp_get_minimum_free_heap_size** (void)

Get the minimum heap that has ever been available.

Return Minimum free heap ever available

uint32_t **esp_random** (void)

Get one random 32-bit word from hardware RNG.

The hardware RNG is fully functional whenever an RF subsystem is running (ie Bluetooth or WiFi is enabled). For random values, call this function after WiFi or Bluetooth are started.

If the RF subsystem is not used by the program, the function `bootloader_random_enable()` can be called to enable an entropy source. `bootloader_random_disable()` must be called before RF subsystem or I2S peripheral are used. See these functions' documentation for more details.

Any time the app is running without an RF subsystem (or `bootloader_random`) enabled, RNG hardware should be considered a PRNG. A very small amount of entropy is available due to pre-seeding while the IDF bootloader is running, but this should not be relied upon for any use.

Return Random value between 0 and `UINT32_MAX`

void **esp_fill_random** (void *buf, size_t len)

Fill a buffer with random bytes from hardware RNG.

Note This function has the same restrictions regarding available entropy as `esp_random()`

Parameters

- `buf`: Pointer to buffer to fill with random numbers.
- `len`: Length of buffer in bytes

esp_err_t **esp_base_mac_addr_set** (const uint8_t *mac)

Set base MAC address with the MAC address which is stored in BLK3 of EFUSE or external storage e.g. flash and EEPROM.

Base MAC address is used to generate the MAC addresses used by the networking interfaces. If using base MAC address stored in BLK3 of EFUSE or external storage, call this API to set base MAC address with the MAC address which is stored in BLK3 of EFUSE or external storage before initializing WiFi/BT/Ethernet.

Note Base MAC must be a unicast MAC (least significant bit of first byte must be zero).

Note If not using a valid OUI, set the “locally administered” bit (bit value 0x02 in the first byte) to avoid collisions.

Return `ESP_OK` on success `ESP_ERR_INVALID_ARG` If `mac` is NULL or is not a unicast MAC

Parameters

- `mac`: base MAC address, length: 6 bytes.

esp_err_t **esp_base_mac_addr_get** (uint8_t *mac)

Return base MAC address which is set using `esp_base_mac_addr_set`.

Return `ESP_OK` on success `ESP_ERR_INVALID_MAC` base MAC address has not been set

Parameters

- `mac`: base MAC address, length: 6 bytes.

esp_err_t **esp_efuse_mac_get_custom** (uint8_t *mac)

Return base MAC address which was previously written to BLK3 of EFUSE.

Base MAC address is used to generate the MAC addresses used by the networking interfaces. This API returns the custom base MAC address which was previously written to BLK3 of EFUSE. Writing this EFUSE allows setting of a different (non-Espressif) base MAC address. It is also possible to store a custom base MAC address elsewhere, see `esp_base_mac_addr_set()` for details.

Return `ESP_OK` on success `ESP_ERR_INVALID_VERSION` An invalid MAC version field was read from BLK3 of EFUSE `ESP_ERR_INVALID_CRC` An invalid MAC CRC was read from BLK3 of EFUSE

Parameters

- `mac`: base MAC address, length: 6 bytes.

esp_err_t **esp_efuse_mac_get_default** (uint8_t *mac)

Return base MAC address which is factory-programmed by Espressif in BLK0 of EFUSE.

Return `ESP_OK` on success

Parameters

- `mac`: base MAC address, length: 6 bytes.

esp_err_t **esp_read_mac** (uint8_t *mac, esp_mac_type_t type)

Read base MAC address and set MAC address of the interface.

This function first get base MAC address using `esp_base_mac_addr_get` or reads base MAC address from BLK0 of EFUSE. Then set the MAC address of the interface including wifi station, wifi softap, bluetooth and ethernet.

Return ESP_OK on success

Parameters

- `mac`: MAC address of the interface, length: 6 bytes.
- `type`: type of MAC address, 0:wifi station, 1:wifi softap, 2:bluetooth, 3:ethernet.

esp_err_t **esp_derive_local_mac** (uint8_t **local_mac*, const uint8_t **universal_mac*)

Derive local MAC address from universal MAC address.

This function derives a local MAC address from an universal MAC address. A definition of local vs universal MAC address can be found on Wikipedia <>. In ESP32, universal MAC address is generated from base MAC address in EFUSE or other external storage. Local MAC address is derived from the universal MAC address.

Return ESP_OK on success

Parameters

- `local_mac`: Derived local MAC address, length: 6 bytes.
- `universal_mac`: Source universal MAC address, length: 6 bytes.

void **esp_chip_info** (*esp_chip_info_t* **out_info*)

Fill an *esp_chip_info_t* structure with information about the chip.

Parameters

- [out] `out_info`: structure to be filled

Structures

struct esp_chip_info_t

The structure represents information about the chip.

Public Members

esp_chip_model_t **model**

chip model, one of *esp_chip_model_t*

uint32_t **features**

bit mask of CHIP_FEATURE_x feature flags

uint8_t **cores**

number of CPU cores

uint8_t **revision**

chip revision number

Macros

CHIP_FEATURE_EMB_FLASH

Chip has embedded flash memory.

CHIP_FEATURE_WIFI_BGN

Chip has 2.4GHz WiFi.

CHIP_FEATURE_BLE

Chip has Bluetooth LE.

CHIP_FEATURE_BT

Chip has Bluetooth Classic.

Type Definitions

typedef void (***shutdown_handler_t**) (void)

Shutdown handler type

Enumerations

enum esp_mac_type_t

Values:

ESP_MAC_WIFI_STA

ESP_MAC_WIFI_SOFTAP

ESP_MAC_BT

ESP_MAC_ETH

enum esp_reset_reason_t

Reset reasons.

Values:

ESP_RST_UNKNOWN

Reset reason can not be determined.

ESP_RST_POWERON

Reset due to power-on event.

ESP_RST_EXT

Reset by external pin (not applicable for ESP32)

ESP_RST_SW

Software reset via esp_restart.

ESP_RST_PANIC

Software reset due to exception/panic.

ESP_RST_INT_WDT

Reset (software or hardware) due to interrupt watchdog.

ESP_RST_TASK_WDT

Reset due to task watchdog.

ESP_RST_WDT

Reset due to other watchdogs.

ESP_RST_DEEPSLEEP

Reset after exiting deep sleep mode.

ESP_RST_BROWNOUT

Brownout reset (software or hardware)

ESP_RST_SDIO

Reset over SDIO.

enum esp_chip_model_t

Chip models.

Values:

CHIP_ESP32 = 1

ESP32.

CHIP_ESP32S2BETA = 2

ESP32-S2 Beta.

Header File

- [esp_common/include/esp_idf_version.h](#)

Functions

const char ***esp_get_idf_version**(void)

Return full IDF version string, same as ‘git describe’ output.

Note If you are printing the ESP-IDF version in a log file or other information, this function provides more information than using the numerical version macros. For example, numerical version macros don’t differentiate between development, pre-release and release versions, but the output of this function does.

Return constant string from `IDF_VER`

Macros

ESP_IDF_VERSION_MAJOR

Major version number (X.x.x)

ESP_IDF_VERSION_MINOR

Minor version number (x.X.x)

ESP_IDF_VERSION_PATCH

Patch version number (x.x.X)

ESP_IDF_VERSION_VAL(major, minor, patch)

Macro to convert IDF version number into an integer

To be used in comparisons, such as `ESP_IDF_VERSION >= ESP_IDF_VERSION_VAL(4, 0, 0)`

ESP_IDF_VERSION

Current IDF version, as an integer

To be used in comparisons, such as `ESP_IDF_VERSION >= ESP_IDF_VERSION_VAL(4, 0, 0)`

2.7.19 Over The Air Updates (OTA)

OTA Process Overview

The OTA update mechanism allows a device to update itself based on data received while the normal firmware is running (for example, over WiFi or Bluetooth.)

OTA requires configuring the *Partition Table* of the device with at least two “OTA app slot” partitions (ie `ota_0` and `ota_1`) and an “OTA Data Partition” .

The OTA operation functions write a new app firmware image to whichever OTA app slot is not currently being used for booting. Once the image is verified, the OTA Data partition is updated to specify that this image should be used for the next boot.

OTA Data Partition

An OTA data partition (type `data`, subtype `ota`) must be included in the *Partition Table* of any project which uses the OTA functions.

For factory boot settings, the OTA data partition should contain no data (all bytes erased to `0xFF`). In this case the esp-idf software bootloader will boot the factory app if it is present in the the partition table. If no factory app is included in the partition table, the first available OTA slot (usually `ota_0`) is booted.

After the first OTA update, the OTA data partition is updated to specify which OTA app slot partition should be booted next.

The OTA data partition is two flash sectors (`0x2000` bytes) in size, to prevent problems if there is a power failure while it is being written. Sectors are independently erased and written with matching data, and if they disagree a counter field is used to determine which sector was written more recently.

App rollback

The main purpose of the application rollback is to keep the device working after the update. This feature allows you to roll back to the previous working application in case a new application has critical errors. When the rollback process is enabled and an OTA update provides a new version of the app, one of three things can happen:

- The application works fine, `esp_ota_mark_app_valid_cancel_rollback()` marks the running application with the state `ESP_OTA_IMG_VALID`. There are no restrictions on booting this application.
- The application has critical errors and further work is not possible, a rollback to the previous application is required, `esp_ota_mark_app_invalid_rollback_and_reboot()` marks the running application with the state `ESP_OTA_IMG_INVALID` and reset. This application will not be selected by the bootloader for boot and will boot the previously working application.
- If the `CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE` option is set, and a reset occurs without calling either function then the application is rolled back.

Note: The state is not written to the binary image of the application it is written to the `otadata` partition. The partition contains a `ota_seq` counter which is a pointer to the slot (`ota_0`, `ota_1`, ...) from which the application will be selected for boot.

App OTA State States control the process of selecting a boot app:

| States | Restriction of selecting a boot app in bootloader |
|---|---|
| <code>ESP_OTA_IMG_VALID</code> | No restriction. Will be selected. |
| <code>ESP_OTA_IMG_UNDEFINED</code> | No restriction. Will be selected. |
| <code>ESP_OTA_IMG_INVALID</code> | Will not be selected. |
| <code>ESP_OTA_IMG_ABORTED</code> | Will not be selected. |
| <code>ESP_OTA_IMG_NEW</code> | If <code>CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE</code> option is set it will be selected only once. In bootloader the state immediately changes to <code>ESP_OTA_IMG_PENDING_VERIFY</code> . |
| <code>ESP_OTA_IMG_PENDING_VERIFY</code> | If <code>CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE</code> option is set it will not be selected and the state will change to <code>ESP_OTA_IMG_ABORTED</code> . |

If `CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE` option is not enabled (by default), then the use of the following functions `esp_ota_mark_app_valid_cancel_rollback()` and `esp_ota_mark_app_invalid_rollback_and_reboot()` are optional, and `ESP_OTA_IMG_NEW` and `ESP_OTA_IMG_PENDING_VERIFY` states are not used.

An option in Kconfig `CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE` allows you to track the first boot of a new application. In this case, the application must confirm its operability by calling `esp_ota_mark_app_valid_cancel_rollback()` function, otherwise the application will be rolled back upon reboot. It allows you to control the operability of the application during the boot phase. Thus, a new application has only one attempt to boot successfully.

Rollback Process The description of the rollback process when `CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE` option is enabled:

- The new application successfully downloaded and `esp_ota_set_boot_partition()` function makes this partition bootable and sets the state `ESP_OTA_IMG_NEW`. This state means that the application is new and should be monitored for its first boot.
- Reboot `esp_restart()`.
- The bootloader checks for the `ESP_OTA_IMG_PENDING_VERIFY` state if it is set, then it will be written to `ESP_OTA_IMG_ABORTED`.
- The bootloader selects a new application to boot so that the state is not set as `ESP_OTA_IMG_INVALID` or `ESP_OTA_IMG_ABORTED`.
- The bootloader checks the selected application for `ESP_OTA_IMG_NEW` state if it is set, then it will be written to `ESP_OTA_IMG_PENDING_VERIFY`. This state means that the application requires confirmation of its operability, if this does not happen and a reboot occurs, this state will be overwritten to

ESP_OTA_IMG_ABORTED (see above) and this application will no longer be able to start, i.e. there will be a rollback to the previous work application.

- A new application has started and should make a self-test.
- If the self-test has completed successfully, then you must call the function `esp_ota_mark_app_valid_cancel_rollback()` because the application is awaiting confirmation of operability (ESP_OTA_IMG_PENDING_VERIFY state).
- If the self-test fails then call `esp_ota_mark_app_invalid_rollback_and_reboot()` function to roll back to the previous working application, while the invalid application is set ESP_OTA_IMG_INVALID state.
- If the application has not been confirmed, the state remains ESP_OTA_IMG_PENDING_VERIFY, and the next boot it will be changed to ESP_OTA_IMG_ABORTED. That will prevent re-boot of this application. There will be a rollback to the previous working application.

Unexpected Reset If a power loss or an unexpected crash occurs at the time of the first boot of a new application, it will roll back the application.

Recommendation: Perform the self-test procedure as quickly as possible, to prevent rollback due to power loss.

Only OTA partitions can be rolled back. Factory partition is not rolled back.

Booting invalid/aborted apps Booting an application which was previously set to ESP_OTA_IMG_INVALID or ESP_OTA_IMG_ABORTED is possible:

- Get the last invalid application partition `esp_ota_get_last_invalid_partition()`.
- Pass the received partition to `esp_ota_set_boot_partition()`, this will update the otadata.
- Restart `esp_restart()`. The bootloader will boot the specified application.

To determine if self-tests should be run during startup of an application, call the `esp_ota_get_state_partition()` function. If result is ESP_OTA_IMG_PENDING_VERIFY then self-testing and subsequent confirmation of operability is required.

Where the states are set A brief description of where the states are set:

- ESP_OTA_IMG_VALID state is set by `esp_ota_mark_app_valid_cancel_rollback()` function.
- ESP_OTA_IMG_UNDEFINED state is set by `esp_ota_set_boot_partition()` function if `CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE` option is not enabled.
- ESP_OTA_IMG_NEW state is set by `esp_ota_set_boot_partition()` function if `CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE` option is enabled.
- ESP_OTA_IMG_INVALID state is set by `esp_ota_mark_app_invalid_rollback_and_reboot()` function.
- ESP_OTA_IMG_ABORTED state is set if there was no confirmation of the application operability and occurs reboots (if `CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE` option is enabled).
- ESP_OTA_IMG_PENDING_VERIFY state is set in a bootloader if `CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE` option is enabled and selected app has ESP_OTA_IMG_NEW state.

Anti-rollback

Anti-rollback prevents rollback to application with security version lower than one programmed in eFuse of chip.

This function works if set `CONFIG_BOOTLOADER_APP_ANTI_ROLLBACK` option. In the bootloader, when selecting a bootable application, an additional security version check is added which is on the chip and in the application image. The version in the bootable firmware must be greater than or equal to the version in the chip.

`CONFIG_BOOTLOADER_APP_ANTI_ROLLBACK` and `CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE` options are used together. In this case, rollback is possible only on the security version which is equal or higher than the version in the chip.

A typical anti-rollback scheme is

- New firmware released with the elimination of vulnerabilities with the previous version of security.
- After the developer makes sure that this firmware is working. He can increase the security version and release a new firmware.
- Download new application.
- To make it bootable, run the function `esp_ota_set_boot_partition()`. If the security version of the new application is smaller than the version in the chip, the new application will be erased. Update to new firmware is not possible.
- Reboot.
- In the bootloader, an application with a security version greater than or equal to the version in the chip will be selected. If otadata is in the initial state, and one firmware was loaded via a serial channel, whose secure version is higher than the chip, then the secure version of efuse will be immediately updated in the bootloader.
- New application booted. Then the application should perform diagnostics of the operation and if it is completed successfully, you should call `esp_ota_mark_app_valid_cancel_rollback()` function to mark the running application with the `ESP_OTA_IMG_VALID` state and update the secure version on chip. Note that if was called `esp_ota_mark_app_invalid_rollback_and_reboot()` function a rollback may not happen due to the device may not have any bootable apps then it will return `ESP_ERR_OTA_ROLLBACK_FAILED` error and stay in the `ESP_OTA_IMG_PENDING_VERIFY` state.
- The next update of app is possible if a running app is in the `ESP_OTA_IMG_VALID` state.

Recommendation:

If you want to avoid the download/erase overhead in case of the app from the server has security version lower than running app you have to get `new_app_info.secure_version` from the first package of an image and compare it with the secure version of efuse. Use `esp_efuse_check_secure_version(new_app_info.secure_version)` function if it is true then continue downloading otherwise abort.

```

....
bool image_header_was_checked = false;
while (1) {
    int data_read = esp_http_client_read(client, ota_write_data, BUFSIZE);
    ...
    if (data_read > 0) {
        if (image_header_was_checked == false) {
            esp_app_desc_t new_app_info;
            if (data_read > sizeof(esp_image_header_t) + sizeof(esp_image_segment_
↪header_t) + sizeof(esp_app_desc_t)) {
                // check current version with downloading
                if (esp_efuse_check_secure_version(new_app_info.secure_version) ==_
↪false) {
                    ESP_LOGE(TAG, "This a new app can not be downloaded due to a_
↪secure version is lower than stored in efuse.");
                    http_cleanup(client);
                    task_fatal_error();
                }

                image_header_was_checked = true;

                esp_ota_begin(update_partition, OTA_SIZE_UNKNOWN, &update_handle);
            }
        }
        esp_ota_write(update_handle, (const void *)ota_write_data, data_read);
    }
}
....

```

Restrictions:

- The number of bits in the `secure_version` field is limited to 32 bits. This means that only 32 times you can do an anti-rollback. You can reduce the length of this efuse field use `CONFIG_BOOTLOADER_APP_SEC_VER_SIZE_EFUSE_FIELD` option.
- Anti-rollback only works if the encoding scheme for efuse is set to NONE.

- The partition table should not have a factory partition, only two of the app.

security_version:

- In application image it is stored in `esp_app_desc` structure. The number is set [CONFIG_BOOTLOADER_APP_SECURE_VERSION](#).
- In ESP32 it is stored in eFuse `EFUSE_BLK3_RDATA4_REG`. (when a eFuse bit is programmed to 1, it can never be reverted to 0). The number of bits set in this register is the `security_version` from app.

Secure OTA Updates Without Secure boot

The verification of signed OTA updates can be performed even without enabling hardware secure boot. For doing so, refer [Signed App Verification Without Hardware Secure Boot](#)

OTA Tool (otatool.py)

The component `app_update` provides a tool [otatool.py](#) for performing OTA partition-related operations on a target device. The following operations can be performed using the tool:

- read contents of otadata partition (`read_otadata`)
- erase otadata partition, effectively resetting device to factory app (`erase_otadata`)
- switch OTA partitions (`switch_ota_partition`)
- erasing OTA partition (`erase_ota_partition`)
- write to OTA partition (`write_ota_partition`)
- read contents of OTA partition (`read_ota_partition`)

The tool can either be imported and used from another Python script or invoked from shell script for users wanting to perform operation programmatically. This is facilitated by the tool's Python API and command-line interface, respectively.

Python API Before anything else, make sure that the `otatool` module is imported.

```
import sys
import os

idf_path = os.environ["IDF_PATH"] # get value of IDF_PATH from environment
otatool_dir = os.path.join(idf_path, "components", "app_update") # otatool.py_
↳ lives in $IDF_PATH/components/app_update

sys.path.append(otatool_dir) # this enables Python to find otatool module
from otatool import * # import all names inside otatool module
```

The starting point for using the tool's Python API to do is create a `OtatoolTarget` object:

```
# Create a partool.py target device connected on serial port /dev/ttyUSB1
target = OtatoolTarget("/dev/ttyUSB1")
```

The created object can now be used to perform operations on the target device:

```
# Erase otadata, resetting the device to factory app
target.erase_otadata()

# Erase contents of OTA app slot 0
target.erase_ota_partition(0)

# Switch boot partition to that of app slot 1
target.switch_ota_partition(1)

# Read OTA partition 'ota_3' and save contents to a file named 'ota_3.bin'
target.read_ota_partition("ota_3", "ota_3.bin")
```

The OTA partition to operate on is specified using either the app slot number or the partition name.

More information on the Python API is available in the docstrings for the tool.

Command-line Interface The command-line interface of *otatool.py* has the following structure:

```
otatool.py [command-args] [subcommand] [subcommand-args]

- command-args - these are arguments that are needed for executing the main_
↳command (parttool.py), mostly pertaining to the target device
- subcommand - this is the operation to be performed
- subcommand-args - these are arguments that are specific to the chosen operation
```

```
# Erase otadata, resetting the device to factory app
otatool.py --port "/dev/ttyUSB1" erase_otadata

# Erase contents of OTA app slot 0
otatool.py --port "/dev/ttyUSB1" erase_ota_partition --slot 0

# Switch boot partition to that of app slot 1
otatool.py --port "/dev/ttyUSB1" switch_ota_partition --slot 1

# Read OTA partition 'ota_3' and save contents to a file named 'ota_3.bin'
otatool.py --port "/dev/ttyUSB1" read_ota_partition --name=ota_3 --output=ota_3.bin
```

More information can be obtained by specifying *-help* as argument:

```
# Display possible subcommands and show main command argument descriptions
otatool.py --help

# Show descriptions for specific subcommand arguments
otatool.py [subcommand] --help
```

See also

- [Partition Table documentation](#)
- [Lower-Level SPI Flash/Partition API](#)
- [ESP HTTPS OTA](#)

Application Example

End-to-end example of OTA firmware update workflow: [system/ota](#).

API Reference

Header File

- [app_update/include/esp_ota_ops.h](#)

Functions

const *esp_app_desc_t* ***esp_ota_get_app_description** (void)

Return *esp_app_desc* structure. This structure includes app version.

Return description for running app.

Return Pointer to *esp_app_desc* structure.

int **esp_ota_get_app_elf_sha256** (char *dst, size_t size)

Fill the provided buffer with SHA256 of the ELF file, formatted as hexadecimal, null-terminated. If the buffer size is not sufficient to fit the entire SHA256 in hex plus a null terminator, the largest possible number of bytes will be written followed by a null.

Return Number of bytes written to dst (including null terminator)

Parameters

- dst: Destination buffer
- size: Size of the buffer

esp_err_t **esp_ota_begin** (const *esp_partition_t* *partition, size_t image_size, *esp_ota_handle_t* *out_handle)

Commence an OTA update writing to the specified partition.

The specified partition is erased to the specified image size.

If image size is not yet known, pass OTA_SIZE_UNKNOWN which will cause the entire partition to be erased.

On success, this function allocates memory that remains in use until esp_ota_end() is called with the returned handle.

Note: If the rollback option is enabled and the running application has the ESP_OTA_IMG_PENDING_VERIFY state then it will lead to the ESP_ERR_OTA_ROLLBACK_INVALID_STATE error. Confirm the running app before to run download a new app, use esp_ota_mark_app_valid_cancel_rollback() function for it (this should be done as early as possible when you first download a new application).

Return

- ESP_OK: OTA operation commenced successfully.
- ESP_ERR_INVALID_ARG: partition or out_handle arguments were NULL, or partition doesn't point to an OTA app partition.
- ESP_ERR_NO_MEM: Cannot allocate memory for OTA operation.
- ESP_ERR_OTA_PARTITION_CONFLICT: Partition holds the currently running firmware, cannot update in place.
- ESP_ERR_NOT_FOUND: Partition argument not found in partition table.
- ESP_ERR_OTA_SELECT_INFO_INVALID: The OTA data partition contains invalid data.
- ESP_ERR_INVALID_SIZE: Partition doesn't fit in configured flash size.
- ESP_ERR_FLASH_OP_TIMEOUT or ESP_ERR_FLASH_OP_FAIL: Flash write failed.
- ESP_ERR_OTA_ROLLBACK_INVALID_STATE: If the running app has not confirmed state. Before performing an update, the application must be valid.

Parameters

- partition: Pointer to info for partition which will receive the OTA update. Required.
- image_size: Size of new OTA app image. Partition will be erased in order to receive this size of image. If 0 or OTA_SIZE_UNKNOWN, the entire partition is erased.
- out_handle: On success, returns a handle which should be used for subsequent esp_ota_write() and esp_ota_end() calls.

esp_err_t **esp_ota_write** (*esp_ota_handle_t* handle, const void *data, size_t size)

Write OTA update data to partition.

This function can be called multiple times as data is received during the OTA operation. Data is written sequentially to the partition.

Return

- ESP_OK: Data was written to flash successfully.
- ESP_ERR_INVALID_ARG: handle is invalid.
- ESP_ERR_OTA_VALIDATE_FAILED: First byte of image contains invalid app image magic byte.
- ESP_ERR_FLASH_OP_TIMEOUT or ESP_ERR_FLASH_OP_FAIL: Flash write failed.
- ESP_ERR_OTA_SELECT_INFO_INVALID: OTA data partition has invalid contents

Parameters

- handle: Handle obtained from esp_ota_begin
- data: Data buffer to write
- size: Size of data buffer in bytes.

esp_err_t **esp_ota_end** (*esp_ota_handle_t* handle)

Finish OTA update and validate newly written app image.

Note After calling `esp_ota_end()`, the handle is no longer valid and any memory associated with it is freed (regardless of result).

Return

- `ESP_OK`: Newly written OTA app image is valid.
- `ESP_ERR_NOT_FOUND`: OTA handle was not found.
- `ESP_ERR_INVALID_ARG`: Handle was never written to.
- `ESP_ERR_OTA_VALIDATE_FAILED`: OTA image is invalid (either not a valid app image, or - if secure boot is enabled - signature failed to verify.)
- `ESP_ERR_INVALID_STATE`: If flash encryption is enabled, this result indicates an internal error writing the final encrypted bytes to flash.

Parameters

- `handle`: Handle obtained from `esp_ota_begin()`.

esp_err_t **esp_ota_set_boot_partition** (**const** *esp_partition_t* *partition)

Configure OTA data for a new boot partition.

Note If this function returns `ESP_OK`, calling `esp_restart()` will boot the newly configured app partition.

Return

- `ESP_OK`: OTA data updated, next reboot will use specified partition.
- `ESP_ERR_INVALID_ARG`: partition argument was NULL or didn't point to a valid OTA partition of type "app".
- `ESP_ERR_OTA_VALIDATE_FAILED`: Partition contained invalid app image. Also returned if secure boot is enabled and signature validation failed.
- `ESP_ERR_NOT_FOUND`: OTA data partition not found.
- `ESP_ERR_FLASH_OP_TIMEOUT` or `ESP_ERR_FLASH_OP_FAIL`: Flash erase or write failed.

Parameters

- `partition`: Pointer to info for partition containing app image to boot.

const *esp_partition_t* ***esp_ota_get_boot_partition** (void)

Get partition info of currently configured boot app.

If `esp_ota_set_boot_partition()` has been called, the partition which was set by that function will be returned.

If `esp_ota_set_boot_partition()` has not been called, the result is usually the same as `esp_ota_get_running_partition()`. The two results are not equal if the configured boot partition does not contain a valid app (meaning that the running partition will be an app that the bootloader chose via fallback).

If the OTA data partition is not present or not valid then the result is the first app partition found in the partition table. In priority order, this means: the factory app, the first OTA app slot, or the test app partition.

Note that there is no guarantee the returned partition is a valid app. Use `esp_image_verify(ESP_IMAGE_VERIFY, ...)` to verify if the returned partition contains a bootable image.

Return Pointer to info for partition structure, or NULL if partition table is invalid or a flash read operation failed. Any returned pointer is valid for the lifetime of the application.

const *esp_partition_t* ***esp_ota_get_running_partition** (void)

Get partition info of currently running app.

This function is different to `esp_ota_get_boot_partition()` in that it ignores any change of selected boot partition caused by `esp_ota_set_boot_partition()`. Only the app whose code is currently running will have its partition information returned.

The partition returned by this function may also differ from `esp_ota_get_boot_partition()` if the configured boot partition is somehow invalid, and the bootloader fell back to a different app partition at boot.

Return Pointer to info for partition structure, or NULL if no partition is found or flash read operation failed. Returned pointer is valid for the lifetime of the application.

const *esp_partition_t* ***esp_ota_get_next_update_partition** (**const** *esp_partition_t* *start_from)

Return the next OTA app partition which should be written with a new firmware.

Call this function to find an OTA app partition which can be passed to esp_ota_begin().

Finds next partition round-robin, starting from the current running partition.

Return Pointer to info for partition which should be updated next. NULL result indicates invalid OTA data partition, or that no eligible OTA app slot partition was found.

Parameters

- start_from: If set, treat this partition info as describing the current running partition. Can be NULL, in which case esp_ota_get_running_partition() is used to find the currently running partition. The result of this function is never the same as this argument.

esp_err_t **esp_ota_get_partition_description** (**const** *esp_partition_t* *partition, *esp_app_desc_t* *app_desc)

Returns esp_app_desc structure for app partition. This structure includes app version.

Returns a description for the requested app partition.

Return

- ESP_OK Successful.
- ESP_ERR_NOT_FOUND app_desc structure is not found. Magic word is incorrect.
- ESP_ERR_NOT_SUPPORTED Partition is not application.
- ESP_ERR_INVALID_ARG Arguments is NULL or if partition's offset exceeds partition size.
- ESP_ERR_INVALID_SIZE Read would go out of bounds of the partition.
- or one of error codes from lower-level flash driver.

Parameters

- [in] partition: Pointer to app partition. (only app partition)
- [out] app_desc: Structure of info about app.

esp_err_t **esp_ota_mark_app_valid_cancel_rollback** (void)

This function is called to indicate that the running app is working well.

Return

- ESP_OK: if successful.

esp_err_t **esp_ota_mark_app_invalid_rollback_and_reboot** (void)

This function is called to roll back to the previously workable app with reboot.

If rollback is successful then device will reset else API will return with error code. Checks applications on a flash drive that can be booted in case of rollback. If the flash does not have at least one app (except the running app) then rollback is not possible.

Return

- ESP_FAIL: if not successful.
- ESP_ERR_OTA_ROLLBACK_FAILED: The rollback is not possible due to flash does not have any apps.

const *esp_partition_t* ***esp_ota_get_last_invalid_partition** (void)

Returns last partition with invalid state (ESP_OTA_IMG_INVALID or ESP_OTA_IMG_ABORTED).

Return partition.

esp_err_t **esp_ota_get_state_partition** (**const** *esp_partition_t* *partition, *esp_ota_img_states_t* *ota_state)

Returns state for given partition.

Return

- ESP_OK: Successful.
- ESP_ERR_INVALID_ARG: partition or ota_state arguments were NULL.
- ESP_ERR_NOT_SUPPORTED: partition is not ota.
- ESP_ERR_NOT_FOUND: Partition table does not have otadata or state was not found for given partition.

Parameters

- [in] `partition`: Pointer to partition.
- [out] `ota_state`: state of partition (if this partition has a record in otadata).

`esp_err_t esp_ota_erase_last_boot_app_partition` (void)

Erase previous boot app partition and corresponding otadata select for this partition.

When current app is marked to as valid then you can erase previous app partition.

Return

- `ESP_OK`: Successful, otherwise `ESP_ERR`.

bool `esp_ota_check_rollback_is_possible` (void)

Checks applications on the slots which can be booted in case of rollback.

These applications should be valid (marked in otadata as not `UNDEFINED`, `INVALID` or `ABORTED` and `crc` is good) and be able booted, and `secure_version` of app \geq `secure_version` of efuse (if anti-rollback is enabled).

Return

- `True`: Returns true if the slots have at least one app (except the running app).
- `False`: The rollback is not possible.

Macros

OTA_SIZE_UNKNOWN

Used for `esp_ota_begin()` if new image size is unknown

ESP_ERR_OTA_BASE

Base error code for `ota_ops` api

ESP_ERR_OTA_PARTITION_CONFLICT

Error if request was to write or erase the current running partition

ESP_ERR_OTA_SELECT_INFO_INVALID

Error if OTA data partition contains invalid content

ESP_ERR_OTA_VALIDATE_FAILED

Error if OTA app image is invalid

ESP_ERR_OTA_SMALL_SEC_VER

Error if the firmware has a secure version less than the running firmware.

ESP_ERR_OTA_ROLLBACK_FAILED

Error if flash does not have valid firmware in passive partition and hence rollback is not possible

ESP_ERR_OTA_ROLLBACK_INVALID_STATE

Error if current active firmware is still marked in pending validation state (`ESP_OTA_IMG_PENDING_VERIFY`), essentially first boot of firmware image post upgrade and hence firmware upgrade is not possible

Type Definitions

`typedef uint32_t esp_ota_handle_t`

Opaque handle for an application OTA update.

`esp_ota_begin()` returns a handle which is then used for subsequent calls to `esp_ota_write()` and `esp_ota_end()`.

2.7.20 Performance Monitor

The Performance Monitor component provides APIs to use ESP32 internal performance counters to profile functions and applications.

Application Example

An example which combines performance monitor is provided in `examples/system/perfmon` directory. This example initializes the performance monitor structure and execute them with printing the statistics.

High level API Reference

Header Files

- [perfmon/include/perfmon.h](#)

API Reference

Header File

- [perfmon/include/xtensa_perfmon_access.h](#)

Functions

esp_err_t **xtensa_perfmon_init** (int *id*, uint16_t *select*, uint16_t *mask*, int *kernelcnt*, int *tracelevel*)

Init Performance Monitor.

Initialize performance monitor register with define values

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if one of the arguments is not correct

Parameters

- [in] *id*: performance counter number
- [in] *select*: select value from PMCTRLx register
- [in] *mask*: mask value from PMCTRLx register
- [in] *kernelcnt*: kernelcnt value from PMCTRLx register
- [in] *tracelevel*: tracelevel value from PMCTRLx register

esp_err_t **xtensa_perfmon_reset** (int *id*)

Reset PM counter.

Reset PM counter. Writes 0 to the PMx register.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if *id* out of range

Parameters

- [in] *id*: performance counter number

void **xtensa_perfmon_start** (void)

Start PM counters.

Start all PM counters synchronously. Write 1 to the PGM register

void **xtensa_perfmon_stop** (void)

Stop PM counters.

Stop all PM counters synchronously. Write 0 to the PGM register

uint32_t **xtensa_perfmon_value** (int *id*)

Read PM counter.

Read value of defined PM counter.

Return

- Performance counter value

Parameters

- [in] *id*: performance counter number

esp_err_t **xtensa_perfmon_overflow** (int *id*)

Read PM overflow state.

Read overflow value of defined PM counter.

Return

- ESP_OK if there is no overflow (overflow = 0)

- ESP_FAIL if overflow occurs (overflow = 1)

Parameters

- [in] id: performance counter number

void **xtensa_perfmon_dump** (void)

Dump PM values.

Dump all PM register to the console.

Header File

- perfmon/include/xtensa_perfmon_apis.h

Functions

esp_err_t **xtensa_perfmon_exec** (const *xtensa_perfmon_config_t* *config)

Execute PM.

Execute performance counter for dedicated function with defined parameters

Return

- ESP_OK if no errors
- ESP_ERR_INVALID_ARG if one of the required parameters not defined
- ESP_FAIL - counter overflow

Parameters

- [in] config: pointer to the configuration structure

void **xtensa_perfmon_view_cb** (void *params, uint32_t select, uint32_t mask, uint32_t value)

Dump PM results.

Callback to dump perfmon result to a FILE* stream specified in perfmon_config_t::callback_params. If callback_params is set to NULL, will print to stdout

Parameters

- [in] params: used parameters passed from configuration (callback_params). This parameter expected as FILE* handle, where data will be stored. If this parameter NULL, then data will be stored to the stdout.
- [in] select: select value for current counter
- [in] mask: mask value for current counter
- [in] value: counter value for current counter

Structures

struct xtensa_perfmon_config

Performance monitor configuration structure.

Structure to configure performance counter to measure dedicated function

Public Members

int **repeat_count**

how much times function will be called before the callback will be repeated

float **max_deviation**

Difference between min and max counter number 0..1, 0 - no difference, 1 - not used

void ***call_params**

This pointer will be passed to the call_function as a parameter

void (***call_function**) (void *params)

pointer to the function that have to be called

void (***callback**) (void *params, uint32_t select, uint32_t mask, uint32_t value)

pointer to the function that will be called with result parameters

`void *callback_params`
parameter that will be passed to the callback

`int tracelevel`
trace level for all counters. In case of negative value, the filter will be ignored. If it's ≥ 0 , then the perfmon will count only when interrupt level $>$ tracelevel. It's useful to monitor interrupts.

`uint32_t counters_size`
amount of counter in the list

`const uint32_t *select_mask`
list of the select/mask parameters

Type Definitions

`typedef struct xtensa_perfmon_config xtensa_perfmon_config_t`

Performance monitor configuration structure.

Structure to configure performance counter to measure dedicated function

2.7.21 Power Management

Overview

Power management algorithm included in ESP-IDF can adjust the advanced peripheral bus (APB) frequency, CPU frequency, and put the chip into light sleep mode to run an application at smallest possible power consumption, given the requirements of application components.

Application components can express their requirements by creating and acquiring power management locks.

For example:

- Driver for a peripheral clocked from APB can request the APB frequency to be set to 80 MHz while the peripheral is used.
- RTOS can request the CPU to run at the highest configured frequency while there are tasks ready to run.
- A peripheral driver may need interrupts to be enabled, which means it will have to request disabling light sleep.

Since requesting higher APB or CPU frequencies or disabling light sleep causes higher current consumption, please keep the usage of power management locks by components to a minimum.

Configuration

Power management can be enabled at compile time, using the option `CONFIG_PM_ENABLE`.

Enabling power management features comes at the cost of increased interrupt latency. Extra latency depends on a number of factors, such as the CPU frequency, single/dual core mode, whether or not frequency switch needs to be done. Minimum extra latency is 0.2 us (when the CPU frequency is 240 MHz and frequency scaling is not enabled). Maximum extra latency is 40 us (when frequency scaling is enabled, and a switch from 40 MHz to 80 MHz is performed on interrupt entry).

Dynamic frequency scaling (DFS) and automatic light sleep can be enabled in an application by calling the function `esp_pm_configure()`. Its argument is a structure defining the frequency scaling settings, `esp_pm_config_esp32_t`. In this structure, three fields need to be initialized:

- `max_freq_mhz`: Maximum CPU frequency in MHz, i.e., the frequency used when the `ESP_PM_CPU_FREQ_MAX` lock is acquired. This field will usually be set to `CONFIG_ESP32_DEFAULT_CPU_FREQ_MHZ`.
- `min_freq_mhz`: Minimum CPU frequency in MHz, i.e., the frequency used when only the `ESP_PM_APB_FREQ_MAX` lock is acquired. This field can be set to the XTAL frequency value, or the XTAL frequency divided by an integer. Note that 10 MHz is the lowest frequency at which the default `REF_TICK` clock of 1 MHz can be generated.

- `light_sleep_enable`: Whether the system should automatically enter light sleep when no locks are acquired (true/false).

Alternatively, if you enable the option `CONFIG_PM_DFS_INIT_AUTO` in menuconfig, the maximum CPU frequency will be determined by the `CONFIG_ESP32_DEFAULT_CPU_FREQ_MHZ` setting, and the minimum CPU frequency will be locked to the XTAL frequency.

Note: Automatic light sleep is based on FreeRTOS Tickless Idle functionality. If automatic light sleep is requested while the option `CONFIG_FREERTOS_USE_TICKLESS_IDLE` is not enabled in menuconfig, `esp_pm_configure()` will return the error `ESP_ERR_NOT_SUPPORTED`.

Note: In light sleep, peripherals are clock gated, and interrupts (from GPIOs and internal peripherals) will not be generated. A wakeup source described in the [Sleep Modes](#) documentation can be used to trigger wakeup from the light sleep state. For example, the EXT0 and EXT1 wakeup sources can be used to wake up the chip via a GPIO.

Power Management Locks

Applications have the ability to acquire/release locks in order to control the power management algorithm. When an application acquires a lock, the power management algorithm operation is restricted in a way described below. When the lock is released, such restrictions are removed.

Power management locks have acquire/release counters. If the lock has been acquired a number of times, it needs to be released the same number of times to remove associated restrictions.

ESP32 supports three types of locks described in the table below.

| Lock | Description |
|------------------------------------|---|
| <code>ESP_PM_CPU_FREQ_REQ</code> | Requests CPU frequency to be at the maximum value set with <code>esp_pm_configure()</code> . For ESP32, this value can be set to 80 MHz, 160 MHz, or 240 MHz. |
| <code>ESP_PM_APB_FREQ_REQ</code> | Requests the APB frequency to be at the maximum supported value. For ESP32, this is 80 MHz. |
| <code>ESP_PM_NO_LIGHT_SLEEP</code> | Disables automatic switching to light sleep. |

ESP32 Power Management Algorithm

The table below shows how CPU and APB frequencies will be switched if dynamic frequency scaling is enabled. You can specify the maximum CPU frequency with either `esp_pm_configure()` or `CONFIG_ESP32_DEFAULT_CPU_FREQ_MHZ`.

| Max CPU Frequency Set | Lock Acquisition | CPU and APB Frequencies |
|-----------------------|---|--|
| 240 | Any of ESP_PM_CPU_FREQ_MAX or ESP_PM_APB_FREQ_MAX acquired | CPU: 240 MHz APB: 80 MHz |
| | None | Min values for both frequencies set with <code>esp_pm_configure()</code> |
| 160 | ESP_PM_CPU_FREQ_MAX acquired | CPU: 160 MHz APB: 80 MHz |
| | ESP_PM_CPU_FREQ_MAX acquired, ESP_PM_APB_FREQ_MAX not acquired | CPU: 80 MHz APB: 80 MHz |
| | None | Min values for both frequencies set with <code>esp_pm_configure()</code> |
| 80 | Any of ESP_PM_CPU_FREQ_MAX or ESP_PM_APB_FREQ_MAX acquired | CPU: 80 MHz APB: 80 MHz |
| | None | Min values for both frequencies set with <code>esp_pm_configure()</code> |

If none of the locks are acquired, and light sleep is enabled in a call to `esp_pm_configure()`, the system will go into light sleep mode. The duration of light sleep will be determined by:

- FreeRTOS tasks blocked with finite timeouts
- Timers registered with *High resolution timer* APIs

Light sleep duration will be chosen to wake up the chip before the nearest event (task being unblocked, or timer elapses).

Dynamic Frequency Scaling and Peripheral Drivers

When DFS is enabled, the APB frequency can be changed multiple times within a single RTOS tick. The APB frequency change does not affect the work of some peripherals, while other peripherals may have issues. For example, Timer Group peripheral timers will keep counting, however, the speed at which they count will change proportionally to the APB frequency.

The following peripherals work normally even when the APB frequency is changing:

- **UART**: if REF_TICK is used as a clock source. See `use_ref_tick` member of `uart_config_t`.
- **LEDC**: if REF_TICK is used as a clock source. See `ledc_timer_config()` function.
- **RMT**: if REF_TICK is used as a clock source. Although the driver does not support REF_TICK, this feature can be enabled by clearing the RMT_REF_ALWAYS_ON_CHx bit for the respective channel.

Currently, the following peripheral drivers are aware of DFS and will use the ESP_PM_APB_FREQ_MAX lock for the duration of the transaction:

- SPI master
- I2C

- I2S (If the APPLL clock is used, then it will use the ESP_PM_NO_LIGHT_SLEEP lock)
- SDMMC

The following drivers will hold the ESP_PM_APB_FREQ_MAX lock while the driver is enabled:

- **SPI slave:** between calls to `spi_slave_initialize()` and `spi_slave_free()`.
- **Ethernet:** between calls to `esp_eth_driver_install()` and `esp_eth_driver_uninstall()`.
- **WiFi:** between calls to `esp_wifi_start()` and `esp_wifi_stop()`. If modem sleep is enabled, the lock will be released for the periods of time when radio is disabled.
- **Bluetooth:** between calls to `esp_bt_controller_enable()` and `esp_bt_controller_disable()`. If Bluetooth modem sleep is enabled, the ESP_PM_APB_FREQ_MAX lock will be released for the periods of time when radio is disabled. However the ESP_PM_NO_LIGHT_SLEEP lock will still be held, unless `CONFIG_BTDM_LOW_POWER_CLOCK` option is set to “External 32kHz crystal”.
- **CAN:** between calls to `can_driver_install()` and `can_driver_uninstall()`.

The following peripheral drivers are not aware of DFS yet. Applications need to acquire/release locks themselves, when necessary:

- MCPWM
- PCNT
- Sigma-delta
- Timer group

API Reference

Header File

- `esp_common/include/esp_pm.h`

Functions

`esp_err_t esp_pm_configure(const void *config)`

Set implementation-specific power management configuration.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if the configuration values are not correct
- ESP_ERR_NOT_SUPPORTED if certain combination of values is not supported, or if CONFIG_PM_ENABLE is not enabled in sdkconfig

Parameters

- `config`: pointer to implementation-specific configuration structure (e.g. `esp_pm_config_esp32`)

`esp_err_t esp_pm_lock_create(esp_pm_lock_type_t lock_type, int arg, const char *name, esp_pm_lock_handle_t *out_handle)`

Initialize a lock handle for certain power management parameter.

When lock is created, initially it is not taken. Call `esp_pm_lock_acquire` to take the lock.

This function must not be called from an ISR.

Return

- ESP_OK on success
- ESP_ERR_NO_MEM if the lock structure can not be allocated
- ESP_ERR_INVALID_ARG if `out_handle` is NULL or type argument is not valid
- ESP_ERR_NOT_SUPPORTED if CONFIG_PM_ENABLE is not enabled in sdkconfig

Parameters

- `lock_type`: Power management constraint which the lock should control
- `arg`: argument, value depends on `lock_type`, see `esp_pm_lock_type_t`
- `name`: arbitrary string identifying the lock (e.g. “wifi” or “spi”). Used by the `esp_pm_dump_locks` function to list existing locks. May be set to NULL. If not set to NULL, must point to a string which is valid for the lifetime of the lock.

- [out] `out_handle`: handle returned from this function. Use this handle when calling `esp_pm_lock_delete`, `esp_pm_lock_acquire`, `esp_pm_lock_release`. Must not be NULL.

esp_err_t **esp_pm_lock_acquire** (*esp_pm_lock_handle_t* handle)

Take a power management lock.

Once the lock is taken, power management algorithm will not switch to the mode specified in a call to `esp_pm_lock_create`, or any of the lower power modes (higher numeric values of 'mode').

The lock is recursive, in the sense that if `esp_pm_lock_acquire` is called a number of times, `esp_pm_lock_release` has to be called the same number of times in order to release the lock.

This function may be called from an ISR.

This function is not thread-safe w.r.t. calls to other `esp_pm_lock_*` functions for the same handle.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if the handle is invalid
- ESP_ERR_NOT_SUPPORTED if CONFIG_PM_ENABLE is not enabled in sdkconfig

Parameters

- handle: handle obtained from `esp_pm_lock_create` function

esp_err_t **esp_pm_lock_release** (*esp_pm_lock_handle_t* handle)

Release the lock taken using `esp_pm_lock_acquire`.

Call to this functions removes power management restrictions placed when taking the lock.

Locks are recursive, so if `esp_pm_lock_acquire` is called a number of times, `esp_pm_lock_release` has to be called the same number of times in order to actually release the lock.

This function may be called from an ISR.

This function is not thread-safe w.r.t. calls to other `esp_pm_lock_*` functions for the same handle.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if the handle is invalid
- ESP_ERR_INVALID_STATE if lock is not acquired
- ESP_ERR_NOT_SUPPORTED if CONFIG_PM_ENABLE is not enabled in sdkconfig

Parameters

- handle: handle obtained from `esp_pm_lock_create` function

esp_err_t **esp_pm_lock_delete** (*esp_pm_lock_handle_t* handle)

Delete a lock created using `esp_pm_lock`.

The lock must be released before calling this function.

This function must not be called from an ISR.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if the handle argument is NULL
- ESP_ERR_INVALID_STATE if the lock is still acquired
- ESP_ERR_NOT_SUPPORTED if CONFIG_PM_ENABLE is not enabled in sdkconfig

Parameters

- handle: handle obtained from `esp_pm_lock_create` function

esp_err_t **esp_pm_dump_locks** (FILE *stream)

Dump the list of all locks to stderr

This function dumps debugging information about locks created using `esp_pm_lock_create` to an output stream.

This function must not be called from an ISR. If `esp_pm_lock_acquire/release` are called while this function is running, inconsistent results may be reported.

Return

- ESP_OK on success

- `ESP_ERR_NOT_SUPPORTED` if `CONFIG_PM_ENABLE` is not enabled in `sdkconfig`

Parameters

- `stream`: stream to print information to; use `stdout` or `stderr` to print to the console; use `fmemopen/open_memstream` to print to a string buffer.

Type Definitions

```
typedef struct esp_pm_lock *esp_pm_lock_handle_t
```

Opaque handle to the power management lock.

Enumerations

```
enum esp_pm_lock_type_t
```

Power management constraints.

Values:

`ESP_PM_CPU_FREQ_MAX`

Require CPU frequency to be at the maximum value set via `esp_pm_configure`. Argument is unused and should be set to 0.

`ESP_PM_APB_FREQ_MAX`

Require APB frequency to be at the maximum value supported by the chip. Argument is unused and should be set to 0.

`ESP_PM_NO_LIGHT_SLEEP`

Prevent the system from going into light sleep. Argument is unused and should be set to 0.

Header File

- [esp32/include/esp32/pm.h](#)

Structures

```
struct esp_pm_config_esp32_t
```

Power management config for ESP32.

Pass a pointer to this structure as an argument to `esp_pm_configure` function.

Public Members

int `max_freq_mhz`
Maximum CPU frequency, in MHz

int `min_freq_mhz`
Minimum CPU frequency to use when no locks are taken, in MHz

bool `light_sleep_enable`
Enter light sleep when no locks are taken

2.7.22 Sleep Modes

Overview

ESP32 is capable of light sleep and deep sleep power saving modes.

In light sleep mode, digital peripherals, most of the RAM, and CPUs are clock-gated, and supply voltage is reduced. Upon exit from light sleep, peripherals and CPUs resume operation, their internal state is preserved.

In deep sleep mode, CPUs, most of the RAM, and all the digital peripherals which are clocked from `APB_CLK` are powered off. The only parts of the chip which can still be powered on are: RTC controller, RTC peripherals (including ULP coprocessor), and RTC memories (slow and fast).

Wakeup from deep and light sleep modes can be done using several sources. These sources can be combined, in this case the chip will wake up when any one of the sources is triggered. Wakeup sources can be enabled using `esp_sleep_enable_X_wakeup` APIs and can be disabled using `esp_sleep_disable_wakeup_source()` API. Next section describes these APIs in detail. Wakeup sources can be configured at any moment before entering light or deep sleep mode.

Additionally, the application can force specific powerdown modes for the RTC peripherals and RTC memories using `esp_sleep_pd_config()` API.

Once wakeup sources are configured, application can enter sleep mode using `esp_light_sleep_start()` or `esp_deep_sleep_start()` APIs. At this point the hardware will be configured according to the requested wakeup sources, and RTC controller will either power down or power off the CPUs and digital peripherals.

WiFi/BT and sleep modes

In deep sleep and light sleep modes, wireless peripherals are powered down. Before entering deep sleep or light sleep modes, applications must disable WiFi and BT using appropriate calls (`esp_bluedroid_disable()`, `esp_bt_controller_disable()`, `esp_wifi_stop()`). WiFi and BT connections will not be maintained in deep sleep or light sleep, even if these functions are not called.

If WiFi connection needs to be maintained, enable WiFi modem sleep, and enable automatic light sleep feature (see *Power Management APIs*). This will allow the system to wake up from sleep automatically when required by WiFi driver, thereby maintaining connection to the AP.

Wakeup sources

Timer RTC controller has a built in timer which can be used to wake up the chip after a predefined amount of time. Time is specified at microsecond precision, but the actual resolution depends on the clock source selected for RTC SLOW_CLK. See chapter “Reset and Clock” of the ESP32 Technical Reference Manual for details about RTC clock options.

This wakeup mode doesn't require RTC peripherals or RTC memories to be powered on during sleep.

`esp_sleep_enable_timer_wakeup()` function can be used to enable deep sleep wakeup using a timer.

Touch pad RTC IO module contains logic to trigger wakeup when a touch sensor interrupt occurs. You need to configure the touch pad interrupt before the chip starts deep sleep.

Revisions 0 and 1 of the ESP32 only support this wakeup mode when RTC peripherals are not forced to be powered on (i.e. `ESP_PD_DOMAIN_RTC_PERIPH` should be set to `ESP_PD_OPTION_AUTO`).

`esp_sleep_enable_touchpad_wakeup()` function can be used to enable this wakeup source.

External wakeup (ext0) RTC IO module contains logic to trigger wakeup when one of RTC GPIOs is set to a predefined logic level. RTC IO is part of RTC peripherals power domain, so RTC peripherals will be kept powered on during deep sleep if this wakeup source is requested.

Because RTC IO module is enabled in this mode, internal pullup or pulldown resistors can also be used. They need to be configured by the application using `rtc_gpio_pullup_en()` and `rtc_gpio_pulldown_en()` functions, before calling `esp_sleep_start()`.

In revisions 0 and 1 of the ESP32, this wakeup source is incompatible with ULP and touch wakeup sources.

`esp_sleep_enable_ext0_wakeup()` function can be used to enable this wakeup source.

Warning: After wake up from sleep, IO pad used for wakeup will be configured as RTC IO. Before using this pad as digital GPIO, reconfigure it using `rtc_gpio_deinit(gpio_num)` function.

External wakeup (ext1) RTC controller contains logic to trigger wakeup using multiple RTC GPIOs. One of the two logic functions can be used to trigger wakeup:

- wake up if any of the selected pins is high (ESP_EXT1_WAKEUP_ANY_HIGH)
- wake up if all the selected pins are low (ESP_EXT1_WAKEUP_ALL_LOW)

This wakeup source is implemented by the RTC controller. As such, RTC peripherals and RTC memories can be powered down in this mode. However, if RTC peripherals are powered down, internal pullup and pulldown resistors will be disabled. To use internal pullup or pulldown resistors, request RTC peripherals power domain to be kept on during sleep, and configure pullup/pulldown resistors using `rtc_gpio_` functions, before entering sleep:

```
esp_sleep_pd_config(ESP_PD_DOMAIN_RTC_PERIPH, ESP_PD_OPTION_ON);
gpio_pullup_dis(gpio_num);
gpio_pulldown_en(gpio_num);
```

Warning: After wake up from sleep, IO pad(s) used for wakeup will be configured as RTC IO. Before using these pads as digital GPIOs, reconfigure them using `rtc_gpio_deinit(gpio_num)` function.

`esp_sleep_enable_ext1_wakeup()` function can be used to enable this wakeup source.

ULP coprocessor wakeup ULP coprocessor can run while the chip is in sleep mode, and may be used to poll sensors, monitor ADC or touch sensor values, and wake up the chip when a specific event is detected. ULP coprocessor is part of RTC peripherals power domain, and it runs the program stored in RTC slow memory. RTC slow memory will be powered on during sleep if this wakeup mode is requested. RTC peripherals will be automatically powered on before ULP coprocessor starts running the program; once the program stops running, RTC peripherals are automatically powered down again.

Revisions 0 and 1 of the ESP32 only support this wakeup mode when RTC peripherals are not forced to be powered on (i.e. ESP_PD_DOMAIN_RTC_PERIPH should be set to ESP_PD_OPTION_AUTO).

`esp_sleep_enable_ulp_wakeup()` function can be used to enable this wakeup source.

GPIO wakeup (light sleep only) In addition to EXT0 and EXT1 wakeup sources described above, one more method of wakeup from external inputs is available in light sleep mode. With this wakeup source, each pin can be individually configured to trigger wakeup on high or low level using `gpio_wakeup_enable()` function. Unlike EXT0 and EXT1 wakeup sources, which can only be used with RTC IOs, this wakeup source can be used with any IO (RTC or digital).

`esp_sleep_enable_gpio_wakeup()` function can be used to enable this wakeup source.

UART wakeup (light sleep only) When ESP32 receives UART input from external devices, it is often required to wake up the chip when input data is available. UART peripheral contains a feature which allows waking up the chip from light sleep when a certain number of positive edges on RX pin are seen. This number of positive edges can be set using `uart_set_wakeup_threshold()` function. Note that the character which triggers wakeup (and any characters before it) will not be received by the UART after wakeup. This means that the external device typically needs to send an extra character to the ESP32 to trigger wakeup, before sending the data.

`esp_sleep_enable_uart_wakeup()` function can be used to enable this wakeup source.

Power-down of RTC peripherals and memories

By default, `esp_deep_sleep_start()` and `esp_light_sleep_start()` functions will power down all RTC power domains which are not needed by the enabled wakeup sources. To override this behaviour, `esp_sleep_pd_config()` function is provided.

Note: in revision 0 of the ESP32, RTC fast memory will always be kept enabled in deep sleep, so that the deep sleep stub can run after reset. This can be overridden, if the application doesn't need clean reset behaviour after deep sleep.

If some variables in the program are placed into RTC slow memory (for example, using `RTC_DATA_ATTR` attribute), RTC slow memory will be kept powered on by default. This can be overridden using `esp_sleep_pd_config()` function, if desired.

Entering light sleep

`esp_light_sleep_start()` function can be used to enter light sleep once wakeup sources are configured. It is also possible to go into light sleep with no wakeup sources configured, in this case the chip will be in light sleep mode indefinitely, until external reset is applied.

Entering deep sleep

`esp_deep_sleep_start()` function can be used to enter deep sleep once wakeup sources are configured. It is also possible to go into deep sleep with no wakeup sources configured, in this case the chip will be in deep sleep mode indefinitely, until external reset is applied.

Configuring IOs

Some ESP32 IOs have internal pullups or pulldowns, which are enabled by default. If an external circuit drives this pin in deep sleep mode, current consumption may increase due to current flowing through these pullups and pulldowns.

To isolate a pin, preventing extra current draw, call `rtc_gpio_isolate()` function.

For example, on ESP32-WROVER module, GPIO12 is pulled up externally. GPIO12 also has an internal pull-down in the ESP32 chip. This means that in deep sleep, some current will flow through these external and internal resistors, increasing deep sleep current above the minimal possible value. Add the following code before `esp_deep_sleep_start()` to remove this extra current:

```
rtc_gpio_isolate(GPIO_NUM_12);
```

UART output handling

Before entering sleep mode, `esp_deep_sleep_start()` will flush the contents of UART FIFOs.

When entering light sleep mode using `esp_light_sleep_start()`, UART FIFOs will not be flushed. Instead, UART output will be suspended, and remaining characters in the FIFO will be sent out after wakeup from light sleep.

Checking sleep wakeup cause

`esp_sleep_get_wakeup_cause()` function can be used to check which wakeup source has triggered wakeup from sleep mode.

For touch pad and ext1 wakeup sources, it is possible to identify pin or touch pad which has caused wakeup using `esp_sleep_get_touchpad_wakeup_status()` and `esp_sleep_get_ext1_wakeup_status()` functions.

Disable sleep wakeup source

Previously configured wakeup source can be disabled later using `esp_sleep_disable_wakeup_source()` API. This function deactivates trigger for the given wakeup source. Additionally it can disable all triggers if the argument is `ESP_SLEEP_WAKEUP_ALL`.

Application Example

Implementation of basic functionality of deep sleep is shown in [protocols/sntp](#) example, where ESP module is periodically waken up to retrieve time from NTP server.

More extensive example in [system/deep_sleep](#) illustrates usage of various deep sleep wakeup triggers and ULP coprocessor programming.

API Reference

Header File

- [esp32/include/esp_sleep.h](#)

Functions

esp_err_t **esp_sleep_disable_wakeup_source** (*esp_sleep_source_t* source)

Disable wakeup source.

This function is used to deactivate wake up trigger for source defined as parameter of the function.

See docs/sleep-modes.rst for details.

Note This function does not modify wake up configuration in RTC. It will be performed in `esp_sleep_start` function.

Return

- ESP_OK on success
- ESP_ERR_INVALID_STATE if trigger was not active

Parameters

- source: - number of source to disable of type `esp_sleep_source_t`

esp_err_t **esp_sleep_enable_ulp_wakeup** (void)

Enable wakeup by ULP coprocessor.

Note ULP wakeup source cannot be used when RTC_PERIPH power domain is forced to be powered on (ESP_PD_OPTION_ON) or when ext0 wakeup source is used.

Return

- ESP_OK on success
- ESP_ERR_NOT_SUPPORTED if additional current by touch (CONFIG_ESP32_RTC_EXT_CRYST_ADDIT_CURRENT) is enabled.
- ESP_ERR_INVALID_STATE if ULP co-processor is not enabled or if wakeup triggers conflict

esp_err_t **esp_sleep_enable_timer_wakeup** (*uint64_t* time_in_us)

Enable wakeup by timer.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if value is out of range (TBD)

Parameters

- time_in_us: time before wakeup, in microseconds

esp_err_t **esp_sleep_enable_touchpad_wakeup** (void)

Enable wakeup by touch sensor.

Note Touch wakeup source cannot be used when RTC_PERIPH power domain is forced to be powered on (ESP_PD_OPTION_ON) or when ext0 wakeup source is used.

Note The FSM mode of the touch button should be configured as the timer trigger mode.

Return

- ESP_OK on success
- ESP_ERR_NOT_SUPPORTED if additional current by touch (CONFIG_ESP32_RTC_EXT_CRYST_ADDIT_CURRENT) is enabled.
- ESP_ERR_INVALID_STATE if wakeup triggers conflict

touch_pad_t **esp_sleep_get_touchpad_wakeup_status** (void)

Get the touch pad which caused wakeup.

If wakeup was caused by another source, this function will return TOUCH_PAD_MAX;

Return touch pad which caused wakeup

esp_err_t **esp_sleep_enable_ext0_wakeup** (*gpio_num_t* gpio_num, int level)

Enable wakeup using a pin.

This function uses external wakeup feature of RTC_IO peripheral. It will work only if RTC peripherals are kept on during sleep.

This feature can monitor any pin which is an RTC IO. Once the pin transitions into the state given by level argument, the chip will be woken up.

Note This function does not modify pin configuration. The pin is configured in esp_sleep_start, immediately before entering sleep mode.

Note ext0 wakeup source cannot be used together with touch or ULP wakeup sources.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if the selected GPIO is not an RTC GPIO, or the mode is invalid
- ESP_ERR_INVALID_STATE if wakeup triggers conflict

Parameters

- *gpio_num*: GPIO number used as wakeup source. Only GPIOs which are have RTC functionality can be used: 0,2,4,12-15,25-27,32-39.
- *level*: input level which will trigger wakeup (0=low, 1=high)

esp_err_t **esp_sleep_enable_ext1_wakeup** (uint64_t mask, *esp_sleep_ext1_wakeup_mode_t* mode)

Enable wakeup using multiple pins.

This function uses external wakeup feature of RTC controller. It will work even if RTC peripherals are shut down during sleep.

This feature can monitor any number of pins which are in RTC IOs. Once any of the selected pins goes into the state given by mode argument, the chip will be woken up.

Note This function does not modify pin configuration. The pins are configured in esp_sleep_start, immediately before entering sleep mode.

Note internal pullups and pulldowns don't work when RTC peripherals are shut down. In this case, external resistors need to be added. Alternatively, RTC peripherals (and pullups/pulldowns) may be kept enabled using esp_sleep_pd_config function.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if any of the selected GPIOs is not an RTC GPIO, or mode is invalid

Parameters

- *mask*: bit mask of GPIO numbers which will cause wakeup. Only GPIOs which are have RTC functionality can be used in this bit map: 0,2,4,12-15,25-27,32-39.
- *mode*: select logic function used to determine wakeup condition:
 - ESP_EXT1_WAKEUP_ALL_LOW: wake up when all selected GPIOs are low
 - ESP_EXT1_WAKEUP_ANY_HIGH: wake up when any of the selected GPIOs is high

esp_err_t **esp_sleep_enable_gpio_wakeup** (void)

Enable wakeup from light sleep using GPIOs.

Each GPIO supports wakeup function, which can be triggered on either low level or high level. Unlike EXT0 and EXT1 wakeup sources, this method can be used both for all IOs: RTC IOs and digital IOs. It can only be used to wakeup from light sleep though.

To enable wakeup, first call gpio_wakeup_enable, specifying gpio number and wakeup level, for each GPIO which is used for wakeup. Then call this function to enable wakeup feature.

Note GPIO wakeup source cannot be used together with touch or ULP wakeup sources.

Return

- ESP_OK on success

- ESP_ERR_INVALID_STATE if wakeup triggers conflict

esp_err_t **esp_sleep_enable_uart_wakeup** (int *uart_num*)

Enable wakeup from light sleep using UART.

Use `uart_set_wakeup_threshold` function to configure UART wakeup threshold.

Wakeup from light sleep takes some time, so not every character sent to the UART can be received by the application.

Note ESP32 does not support wakeup from UART2.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if wakeup from given UART is not supported

Parameters

- *uart_num*: UART port to wake up from

uint64_t **esp_sleep_get_ext1_wakeup_status** (void)

Get the bit mask of GPIOs which caused wakeup (*ext1*)

If wakeup was caused by another source, this function will return 0.

Return bit mask, if GPIO*n* caused wakeup, BIT(*n*) will be set

esp_err_t **esp_sleep_pd_config** (*esp_sleep_pd_domain_t* *domain*, *esp_sleep_pd_option_t* *option*)

Set power down mode for an RTC power domain in sleep mode.

If not set using this API, all power domains default to ESP_PD_OPTION_AUTO.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if either of the arguments is out of range

Parameters

- *domain*: power domain to configure
- *option*: power down option (ESP_PD_OPTION_OFF, ESP_PD_OPTION_ON, or ESP_PD_OPTION_AUTO)

void **esp_deep_sleep_start** (void)

Enter deep sleep with the configured wakeup options.

This function does not return.

esp_err_t **esp_light_sleep_start** (void)

Enter light sleep with the configured wakeup options.

Return

- ESP_OK on success (returned after wakeup)
- ESP_ERR_INVALID_STATE if WiFi or BT is not stopped

void **esp_deep_sleep** (uint64_t *time_in_us*)

Enter deep-sleep mode.

The device will automatically wake up after the deep-sleep time. Upon waking up, the device calls deep sleep wake stub, and then proceeds to load application.

Call to this function is equivalent to a call to `esp_deep_sleep_enable_timer_wakeup` followed by a call to `esp_deep_sleep_start`.

`esp_deep_sleep` does not shut down WiFi, BT, and higher level protocol connections gracefully. Make sure relevant WiFi and BT stack functions are called to close any connections and deinitialize the peripherals. These include:

- `esp_bluedroid_disable`
- `esp_bt_controller_disable`
- `esp_wifi_stop`

This function does not return.

Parameters

- `time_in_us`: deep-sleep time, unit: microsecond

`esp_sleep_wakeup_cause_t esp_sleep_get_wakeup_cause` (void)

Get the wakeup source which caused wakeup from sleep.

Return cause of wake up from last sleep (deep sleep or light sleep)

void `esp_wake_deep_sleep` (void)

Default stub to run on wake from deep sleep.

Allows for executing code immediately on wake from sleep, before the software bootloader or ESP-IDF app has started up.

This function is weak-linked, so you can implement your own version to run code immediately when the chip wakes from sleep.

See docs/deep-sleep-stub.rst for details.

void `esp_set_deep_sleep_wake_stub` (`esp_deep_sleep_wake_stub_fn_t new_stub`)

Install a new stub at runtime to run on wake from deep sleep.

If implementing `esp_wake_deep_sleep()` then it is not necessary to call this function.

However, it is possible to call this function to substitute a different deep sleep stub. Any function used as a deep sleep stub must be marked `RTC_IRAM_ATTR`, and must obey the same rules given for `esp_wake_deep_sleep()`.

`esp_deep_sleep_wake_stub_fn_t esp_get_deep_sleep_wake_stub` (void)

Get current wake from deep sleep stub.

Return Return current wake from deep sleep stub, or NULL if no stub is installed.

void `esp_default_wake_deep_sleep` (void)

The default esp-idf-provided `esp_wake_deep_sleep()` stub.

See docs/deep-sleep-stub.rst for details.

void `esp_deep_sleep_disable_rom_logging` (void)

Disable logging from the ROM code after deep sleep.

Using LSB of `RTC_STORE4`.

Type Definitions

`typedef esp_sleep_source_t esp_sleep_wakeup_cause_t`

`typedef void (*esp_deep_sleep_wake_stub_fn_t) (void)`

Function type for stub to run on wake from sleep.

Enumerations

`enum esp_sleep_ext1_wakeup_mode_t`

Logic function used for EXT1 wakeup mode.

Values:

`ESP_EXT1_WAKEUP_ALL_LOW = 0`

Wake the chip when all selected GPIOs go low.

`ESP_EXT1_WAKEUP_ANY_HIGH = 1`

Wake the chip when any of the selected GPIOs go high.

`enum esp_sleep_pd_domain_t`

Power domains which can be powered down in sleep mode.

Values:

`ESP_PD_DOMAIN_RTC_PERIPH`

RTC IO, sensors and ULP co-processor.

ESP_PD_DOMAIN_RTC_SLOW_MEM
RTC slow memory.

ESP_PD_DOMAIN_RTC_FAST_MEM
RTC fast memory.

ESP_PD_DOMAIN_XTAL
XTAL oscillator.

ESP_PD_DOMAIN_MAX
Number of domains.

enum esp_sleep_pd_option_t
Power down options.

Values:

ESP_PD_OPTION_OFF
Power down the power domain in sleep mode.

ESP_PD_OPTION_ON
Keep power domain enabled during sleep mode.

ESP_PD_OPTION_AUTO
Keep power domain enabled in sleep mode, if it is needed by one of the wakeup options. Otherwise power it down.

enum esp_sleep_source_t
Sleep wakeup cause.

Values:

ESP_SLEEP_WAKEUP_UNDEFINED
In case of deep sleep, reset was not caused by exit from deep sleep.

ESP_SLEEP_WAKEUP_ALL
Not a wakeup cause, used to disable all wakeup sources with `esp_sleep_disable_wakeup_source`.

ESP_SLEEP_WAKEUP_EXT0
Wakeup caused by external signal using RTC_IO.

ESP_SLEEP_WAKEUP_EXT1
Wakeup caused by external signal using RTC_CNTL.

ESP_SLEEP_WAKEUP_TIMER
Wakeup caused by timer.

ESP_SLEEP_WAKEUP_TOUCHPAD
Wakeup caused by touchpad.

ESP_SLEEP_WAKEUP_ULP
Wakeup caused by ULP program.

ESP_SLEEP_WAKEUP_GPIO
Wakeup caused by GPIO (light sleep only)

ESP_SLEEP_WAKEUP_UART
Wakeup caused by UART (light sleep only)

2.7.23 Watchdogs

Overview

The ESP-IDF has support for two types of watchdogs: The Interrupt Watchdog Timer and the Task Watchdog Timer (TWDT). The Interrupt Watchdog Timer and the TWDT can both be enabled using [Project Configuration Menu](#), however the TWDT can also be enabled during runtime. The Interrupt Watchdog is responsible for detecting instances

where FreeRTOS task switching is blocked for a prolonged period of time. The TWDT is responsible for detecting instances of tasks running without yielding for a prolonged period.

Interrupt watchdog The interrupt watchdog makes sure the FreeRTOS task switching interrupt isn't blocked for a long time. This is bad because no other tasks, including potentially important ones like the WiFi task and the idle task, can't get any CPU runtime. A blocked task switching interrupt can happen because a program runs into an infinite loop with interrupts disabled or hangs in an interrupt.

The default action of the interrupt watchdog is to invoke the panic handler, causing a register dump and an opportunity for the programmer to find out, using either OpenOCD or gdbstub, what bit of code is stuck with interrupts disabled. Depending on the configuration of the panic handler, it can also blindly reset the CPU, which may be preferred in a production environment.

The interrupt watchdog is built around the hardware watchdog in timer group 1. If this watchdog for some reason cannot execute the NMI handler that invokes the panic handler (e.g. because IRAM is overwritten by garbage), it will hard-reset the SOC.

Task Watchdog Timer The Task Watchdog Timer (TWDT) is responsible for detecting instances of tasks running for a prolonged period of time without yielding. This is a symptom of CPU starvation and is usually caused by a higher priority task looping without yielding to a lower-priority task thus starving the lower priority task from CPU time. This can be an indicator of poorly written code that spinloops on a peripheral, or a task that is stuck in an infinite loop.

By default the TWDT will watch the Idle Tasks of each CPU, however any task can elect to be watched by the TWDT. Each watched task must 'reset' the TWDT periodically to indicate that they have been allocated CPU time. If a task does not reset within the TWDT timeout period, a warning will be printed with information about which tasks failed to reset the TWDT in time and which tasks are currently running on the ESP32 CPUs. And also there is a possibility to redefine the function `esp_task_wdt_isr_user_handler` in the user code to receive this event.

The TWDT is built around the Hardware Watchdog Timer in Timer Group 0. The TWDT can be initialized by calling `esp_task_wdt_init()` which will configure the hardware timer. A task can then subscribe to the TWDT using `esp_task_wdt_add()` in order to be watched. Each subscribed task must periodically call `esp_task_wdt_reset()` to reset the TWDT. Failure by any subscribed tasks to periodically call `esp_task_wdt_reset()` indicates that one or more tasks have been starved of CPU time or are stuck in a loop somewhere.

A watched task can be unsubscribed from the TWDT using `esp_task_wdt_delete()`. A task that has been unsubscribed should no longer call `esp_task_wdt_reset()`. Once all tasks have unsubscribed from the TWDT, the TWDT can be deinitialized by calling `esp_task_wdt_deinit()`.

By default `CONFIG_ESP_TASK_WDT` in *Project Configuration Menu* be enabled causing the TWDT to be initialized automatically during startup. Likewise `CONFIG_ESP_TASK_WDT_CHECK_IDLE_TASK_CPU0` and `CONFIG_ESP_TASK_WDT_CHECK_IDLE_TASK_CPU1` are also enabled by default causing the two Idle Tasks to be subscribed to the TWDT during startup.

JTAG and watchdogs While debugging using OpenOCD, the CPUs will be halted every time a breakpoint is reached. However if the watchdog timers continue to run when a breakpoint is encountered, they will eventually trigger a reset making it very difficult to debug code. Therefore OpenOCD will disable the hardware timers of both the interrupt and task watchdogs at every breakpoint. Moreover, OpenOCD will not reenale them upon leaving the breakpoint. This means that interrupt watchdog and task watchdog functionality will essentially be disabled. No warnings or panics from either watchdogs will be generated when the ESP32 is connected to OpenOCD via JTAG.

Interrupt Watchdog API Reference

Header File

- `esp32/include/esp_int_wdt.h`

Functions

void **esp_int_wdt_init** (void)

Initialize the non-CPU-specific parts of interrupt watchdog. This is called in the init code if the interrupt watchdog is enabled in menuconfig.

Task Watchdog API Reference

A full example using the Task Watchdog is available in esp-idf: [system/task_watchdog](#)

Header File

- [esp_common/include/esp_task_wdt.h](#)

Functions

esp_err_t **esp_task_wdt_init** (uint32_t *timeout*, bool *panic*)

Initialize the Task Watchdog Timer (TWDT)

This function configures and initializes the TWDT. If the TWDT is already initialized when this function is called, this function will update the TWDT's timeout period and panic configurations instead. After initializing the TWDT, any task can elect to be watched by the TWDT by subscribing to it using `esp_task_wdt_add()`.

Return

- ESP_OK: Initialization was successful
- ESP_ERR_NO_MEM: Initialization failed due to lack of memory

Note `esp_task_wdt_init()` must only be called after the scheduler started

Parameters

- [in] *timeout*: Timeout period of TWDT in seconds
- [in] *panic*: Flag that controls whether the panic handler will be executed when the TWDT times out

esp_err_t **esp_task_wdt_deinit** (void)

Deinitialize the Task Watchdog Timer (TWDT)

This function will deinitialize the TWDT. Calling this function whilst tasks are still subscribed to the TWDT, or when the TWDT is already deinitialized, will result in an error code being returned.

Return

- ESP_OK: TWDT successfully deinitialized
- ESP_ERR_INVALID_STATE: Error, tasks are still subscribed to the TWDT
- ESP_ERR_NOT_FOUND: Error, TWDT has already been deinitialized

esp_err_t **esp_task_wdt_add** (*TaskHandle_t* *handle*)

Subscribe a task to the Task Watchdog Timer (TWDT)

This function subscribes a task to the TWDT. Each subscribed task must periodically call `esp_task_wdt_reset()` to prevent the TWDT from elapsing its timeout period. Failure to do so will result in a TWDT timeout. If the task being subscribed is one of the Idle Tasks, this function will automatically enable `esp_task_wdt_reset()` to be called from the Idle Hook of the Idle Task. Calling this function whilst the TWDT is uninitialized or attempting to subscribe an already subscribed task will result in an error code being returned.

Return

- ESP_OK: Successfully subscribed the task to the TWDT
- ESP_ERR_INVALID_ARG: Error, the task is already subscribed
- ESP_ERR_NO_MEM: Error, could not subscribe the task due to lack of memory
- ESP_ERR_INVALID_STATE: Error, the TWDT has not been initialized yet

Parameters

- [in] *handle*: Handle of the task. Input NULL to subscribe the current running task to the TWDT

`esp_err_t esp_task_wdt_reset` (void)

Reset the Task Watchdog Timer (TWDT) on behalf of the currently running task.

This function will reset the TWDT on behalf of the currently running task. Each subscribed task must periodically call this function to prevent the TWDT from timing out. If one or more subscribed tasks fail to reset the TWDT on their own behalf, a TWDT timeout will occur. If the IDLE tasks have been subscribed to the TWDT, they will automatically call this function from their idle hooks. Calling this function from a task that has not subscribed to the TWDT, or when the TWDT is uninitialized will result in an error code being returned.

Return

- ESP_OK: Successfully reset the TWDT on behalf of the currently running task
- ESP_ERR_NOT_FOUND: Error, the current running task has not subscribed to the TWDT
- ESP_ERR_INVALID_STATE: Error, the TWDT has not been initialized yet

`esp_err_t esp_task_wdt_delete` (*TaskHandle_t* handle)

Unsubscribes a task from the Task Watchdog Timer (TWDT)

This function will unsubscribe a task from the TWDT. After being unsubscribed, the task should no longer call `esp_task_wdt_reset()`. If the task is an IDLE task, this function will automatically disable the calling of `esp_task_wdt_reset()` from the Idle Hook. Calling this function whilst the TWDT is uninitialized or attempting to unsubscribe an already unsubscribed task from the TWDT will result in an error code being returned.

Return

- ESP_OK: Successfully unsubscribed the task from the TWDT
- ESP_ERR_INVALID_ARG: Error, the task is already unsubscribed
- ESP_ERR_INVALID_STATE: Error, the TWDT has not been initialized yet

Parameters

- [in] handle: Handle of the task. Input NULL to unsubscribe the current running task.

`esp_err_t esp_task_wdt_status` (*TaskHandle_t* handle)

Query whether a task is subscribed to the Task Watchdog Timer (TWDT)

This function will query whether a task is currently subscribed to the TWDT, or whether the TWDT is initialized.

Return :

- ESP_OK: The task is currently subscribed to the TWDT
- ESP_ERR_NOT_FOUND: The task is currently not subscribed to the TWDT
- ESP_ERR_INVALID_STATE: The TWDT is not initialized, therefore no tasks can be subscribed

Parameters

- [in] handle: Handle of the task. Input NULL to query the current running task.

2.7.24 System Time

Overview

System time can be kept using either one time source or two time sources simultaneously. The choice depends on the application purpose and accuracy requirements for system time.

There are the following two time sources:

- **RTC timer:** Allows keeping the system time during any resets and sleep modes, only the power-up reset leads to resetting the RTC timer. The frequency deviation depends on an *RTC Clock Source* and affects accuracy only in sleep modes, in which case the time will be measured at 6.6667 us resolution.
- **High-resolution timer:** Not available during any reset and sleep modes. The reason for using this timer is to achieve greater accuracy. It uses the APB_CLK clock source (typically 80 MHz), which has a frequency deviation of less than ± 10 ppm. Time will be measured at 1 us resolution.

The settings for the system time source are as follows:

- RTC and high-resolution timer (default)
- RTC

- High-resolution timer
- None

It is recommended to stick to the default setting which provides maximum accuracy. If you want to choose a different timer, configure `CONFIG_ESP32_TIME_SYSCALL` in project configuration.

RTC Clock Source

The RTC timer has the following clock sources:

- Internal 150kHz RC oscillator (default): Features lowest deep sleep current consumption and no dependence on any external components. However, as frequency stability is affected by temperature fluctuations, time may drift in both Deep and Light sleep modes.
- External 32kHz crystal: Requires a 32kHz crystal to be connected to the 32K_XP and 32K_NP pins. Provides better frequency stability at the expense of slightly higher (by 1 uA) Deep sleep current consumption.
- External 32kHz oscillator at 32K_XP pin: Allows using 32kHz clock generated by an external circuit. The external clock signal must be connected to the 32K_XP pin. The amplitude should be less than 1.2 V for sine wave signal and less than 1 V for square wave signal. Common mode voltage should be in the range of $0.1 < V_{cm} < 0.5 \times V_{amp}$, where V_{amp} is signal amplitude. Additionally, a 1 nF capacitor must be placed between the 32K_XN pin and ground. In this case, the 32K_XN pin cannot be used as a GPIO pin.
- Internal 8.5MHz oscillator, divided by 256 (~33kHz). Provides better frequency stability than the internal 150kHz RC oscillator at the expense of higher (by 5 uA) deep sleep current consumption. It also does not require external components.

The choice depends on your requirements for system time accuracy and power consumption in sleep modes. To modify the RTC clock source, set `CONFIG_ESP32_RTC_CLK_SRC` in project configuration.

More details on wiring requirements for the External 32kHz crystal and External 32kHz oscillator at 32K_XP pin sources can be found in Section 2.1.4 *Crystal Oscillator* of [Hardware Design Guidelines](#).

Get Current Time

To get the current time, use the POSIX function `gettimeofday()`. Additionally, you can use the following standard C library functions to obtain time and manipulate it:

```
gettimeofday
time
asctime
clock
ctime
difftime
gmtime
localtime
mktime
strptime
adjtime*
```

* –To stop smooth time adjustment and update the current time immediately, use the POSIX function `settimeofday()`.

If you need to obtain time with one second resolution, use the following method:

```
time_t now;
char strftime_buf[64];
struct tm timeinfo;

time(&now);
// Set timezone to China Standard Time
setenv("TZ", "CST-8", 1);
tzset();
```

(continues on next page)

(continued from previous page)

```
localtime_r(&now, &timeinfo);
strftime(strftime_buf, sizeof(strftime_buf), "%c", &timeinfo);
ESP_LOGI(TAG, "The current date/time in Shanghai is: %s", strftime_buf);
```

If you need to obtain time with one microsecond resolution, use the code snippet below:

```
struct timeval tv_now;
gettimeofday(&tv_now, NULL);
int64_t time_us = (int64_t)tv_now.tv_sec * 1000000L + (int64_t)tv_now.tv_usec;
```

SNTP Time Synchronization

To set the current time, you can use the POSIX functions `settimeofday()` and `adjtime()`. They are used internally in the lwIP SNTP library to set current time when a response from the NTP server is received. These functions can also be used separately from the lwIP SNTP library.

A function to use inside the lwIP SNTP library depends on a sync mode for system time. Use the function `sntp_set_sync_mode()` to set one of the following sync modes:

- `SNTP_SYNC_MODE_IMMED` (default) updates system time immediately upon receiving a response from the SNTP server after using `settimeofday()`.
- `SNTP_SYNC_MODE_SMOOTH` updates time smoothly by gradually reducing time error using the function `adjtime()`. If the difference between the SNTP response time and system time is more than 35 minutes, update system time immediately by using `settimeofday()`.

The lwIP SNTP library has API functions for setting a callback function for a certain event. You might need the following functions:

- `sntp_set_time_sync_notification_cb()` - use it for setting a callback function that will notify of the time synchronization process
- `sntp_get_sync_status()` and `sntp_set_sync_status()` - use it to get/set time synchronization status

To start synchronization via SNTP, just call the following three functions.

```
sntp_setoperatingmode(SNTP_OPMODE_POLL);
sntp_setservername(0, "pool.ntp.org");
sntp_init();
```

An application with this initialization code will periodically synchronize the time. The time synchronization period is determined by `CONFIG_LWIP_SNTP_UPDATE_DELAY` (default value is one hour). To modify the variable, set `CONFIG_LWIP_SNTP_UPDATE_DELAY` in project configuration.

A code example that demonstrates the implementation of time synchronization based on the lwIP SNTP library is provided in `protocols/sntp` directory.

Timezones

To set local timezone, use the following POSIX functions:

1. Call `setenv()` to set the TZ environment variable to the correct value depending on the device location. The format of the time string is the same as described in the [GNU libc documentation](#) (although the implementation is different).
2. Call `tzset()` to update C library runtime data for the new time zone.

Once these steps are completed, call the standard C library function `localtime()`, and it will return correct local time taking into account the time zone offset and daylight saving time.

API Reference

Header File

- [lwip/include/apps/sntp/sntp.h](#)

Functions

void **sntp_sync_time** (**struct** timeval *tv)

This function updates the system time.

This is a weak-linked function. It is possible to replace all SNTP update functionality by placing a `sntp_sync_time()` function in the app firmware source. If the default implementation is used, calling `sntp_set_sync_mode()` allows the time synchronization mode to be changed to instant or smooth. If a callback function is registered via `sntp_set_time_sync_notification_cb()`, it will be called following time synchronization.

Parameters

- tv: Time received from SNTP server.

void **sntp_set_sync_mode** (*sntp_sync_mode_t* sync_mode)

Set the sync mode.

Allowable two mode: `SNTP_SYNC_MODE_IMMED` and `SNTP_SYNC_MODE_SMOOTH`.

Parameters

- sync_mode: Sync mode.

sntp_sync_mode_t **sntp_get_sync_mode** (void)

Get set sync mode.

Return `SNTP_SYNC_MODE_IMMED`: Update time immediately. `SNTP_SYNC_MODE_SMOOTH`: Smooth time updating.

sntp_sync_status_t **sntp_get_sync_status** (void)

Get status of time sync.

After the update is completed, the status will be returned as `SNTP_SYNC_STATUS_COMPLETED`. After that, the status will be reset to `SNTP_SYNC_STATUS_RESET`. If the update operation is not completed yet, the status will be `SNTP_SYNC_STATUS_RESET`. If a smooth mode was chosen and the synchronization is still continuing (adjtime works), then it will be `SNTP_SYNC_STATUS_IN_PROGRESS`.

Return `SNTP_SYNC_STATUS_RESET`: Reset status. `SNTP_SYNC_STATUS_COMPLETED`: Time is synchronized. `SNTP_SYNC_STATUS_IN_PROGRESS`: Smooth time sync in progress.

void **sntp_set_sync_status** (*sntp_sync_status_t* sync_status)

Set status of time sync.

Parameters

- sync_status: status of time sync (see `sntp_sync_status_t`)

void **sntp_set_time_sync_notification_cb** (*sntp_sync_time_cb_t* callback)

Set a callback function for time synchronization notification.

Parameters

- callback: a callback function

void **sntp_set_sync_interval** (uint32_t interval_ms)

Set the sync interval of SNTP operation.

Note: SNTPv4 RFC 4330 enforces a minimum sync interval of 15 seconds. This sync interval will be used in the next attempt update time through SNTP. To apply the new sync interval call the `sntp_restart()` function, otherwise, it will be applied after the last interval expired.

Parameters

- interval_ms: The sync interval in ms. It cannot be lower than 15 seconds, otherwise 15 seconds will be set.

`uint32_t sntp_get_sync_interval` (void)

Get the sync interval of SNTP operation.

Return the sync interval

bool `sntp_restart` (void)

Restart SNTP.

Return True - Restart False - SNTP was not initialized yet

Type Definitions

`typedef void (*sntp_sync_time_cb_t) (struct timeval *tv)`

SNTP callback function for notifying about time sync event.

Parameters

- `tv`: Time received from SNTP server.

Enumerations

`enum sntp_sync_mode_t`

SNTP time update mode.

Values:

`SNTP_SYNC_MODE_IMMED`

Update system time immediately when receiving a response from the SNTP server.

`SNTP_SYNC_MODE_SMOOTH`

Smooth time updating. Time error is gradually reduced using `adjtime` function. If the difference between SNTP response time and system time is large (more than 35 minutes) then update immediately.

`enum sntp_sync_status_t`

SNTP sync status.

Values:

`SNTP_SYNC_STATUS_RESET`

`SNTP_SYNC_STATUS_COMPLETED`

`SNTP_SYNC_STATUS_IN_PROGRESS`

Code examples for this API section are provided in the [system](#) directory of ESP-IDF examples.

2.8 Project Configuration

2.8.1 Introduction

ESP-IDF uses [kconfiglib](#) which is a Python-based extension to the [Kconfig](#) system which provides a compile-time project configuration mechanism. Kconfig is based around options of several types: integer, string, boolean. Kconfig files specify dependencies between options, default values of the options, the way the options are grouped together, etc.

For the complete list of available features please see [Kconfig](#) and [kconfiglib](#) extentions.

2.8.2 Project Configuration Menu

Application developers can open a terminal-based project configuration menu with the `idf.py menuconfig` build target.

After being updated, this configuration is saved inside `sdkconfig` file in the project root directory. Based on `sd-kconfig`, application build targets will generate `sdkconfig.h` file in the build directory, and will make `sdkconfig` options available to the project build system and source files.

(For the legacy GNU Make build system, the project configuration menu is opened with `make menuconfig`.)

2.8.3 Using `sdkconfig.defaults`

In some cases, such as when `sdkconfig` file is under revision control, the fact that `sdkconfig` file gets changed by the build system may be inconvenient. The build system offers a way to avoid this, in the form of `sdkconfig.defaults` file. This file is never touched by the build system, and must be created manually. It can contain all the options which matter for the given application. The format is the same as that of the `sdkconfig` file. Once `sdkconfig.defaults` is created, `sdkconfig` can be deleted and added to the ignore list of the revision control system (e.g. `.gitignore` file for git). Project build targets will automatically create `sdkconfig` file, populated with the settings from `sdkconfig.defaults` file, and the rest of the settings will be set to their default values. Note that the build process will not override settings that are already in `sdkconfig` by ones from `sdkconfig.defaults`. For more information, see [Custom `sdkconfig.defaults`](#).

2.8.4 Kconfig Formatting Rules

The following attributes of `Kconfig` files are standardized:

- Within any menu, option names should have a consistent prefix. The prefix length is currently set to at least 3 characters.
- The indentation style is 4 characters created by spaces. All sub-items belonging to a parent item are indented by one level deeper. For example, `menu` is indented by 0 characters, the `config` inside of the menu by 4 characters, the help of the `config` by 8 characters and the text of the help by 12 characters.
- No trailing spaces are allowed at the end of the lines.
- The maximum length of options is set to 40 characters.
- The maximum length of lines is set to 120 characters.
- Lines cannot be wrapped by backslash (because there is a bug in earlier versions of `conf-idf` which causes that Windows line endings are not recognized after a backslash).

Format checker

`tools/check_kconfigs.py` is provided for checking the `Kconfig` formatting rules. The checker checks all `Kconfig` and `Kconfig.projbuild` files in the ESP-IDF directory and generates a new file with suffix `.new` with some recommendations how to fix issues (if there are any). Please note that the checker cannot correct all rules and the responsibility of the developer is to check and make final corrections in order to pass the tests. For example, indentations will be corrected if there isn't some misleading previous formatting but it cannot come up with a common prefix for options inside a menu.

2.8.5 Backward Compatibility of Kconfig Options

The standard `Kconfig` tools ignore unknown options in `sdkconfig`. So if a developer has custom settings for options which are renamed in newer ESP-IDF releases then the given setting for the option would be silently ignored. Therefore, several features have been adopted to avoid this:

1. `confgen.py` is used by the tool chain to pre-process `sdkconfig` files before anything else, for example `menuconfig`, would read them. As the consequence, the settings for old options will be kept and not ignored.
2. `confgen.py` recursively finds all `sdkconfig.rename` files in ESP-IDF directory which contain old and new `Kconfig` option names. Old options are replaced by new ones in the `sdkconfig` file.
3. `confgen.py` post-processes `sdkconfig` files and generates all build outputs (`sdkconfig.h`, `sdkconfig.cmake`, `auto.conf`) by adding a list of compatibility statements, i.e. value of the old option is set the value of the new option (after modification). This is done in order to not break customer codes where old option might still be used.
4. [Deprecated options and their replacements](#) are automatically generated by `confgen.py`.

2.8.6 Configuration Options Reference

Subsequent sections contain the list of available ESP-IDF options, automatically generated from Kconfig files. Note that depending on the options selected, some options listed here may not be visible by default in the interface of menuconfig.

By convention, all option names are upper case with underscores. When Kconfig generates sdkconfig and sdkconfig.h files, option names are prefixed with `CONFIG_`. So if an option `ENABLE_FOO` is defined in a Kconfig file and selected in menuconfig, then `sdkconfig` and `sdkconfig.h` files will have `CONFIG_ENABLE_FOO` defined. In this reference, option names are also prefixed with `CONFIG_`, same as in the source code.

SDK tool configuration

Contains:

- [*CONFIG_SDK_TOOLPREFIX*](#)
- [*CONFIG_SDK_PYTHON*](#)
- [*CONFIG_SDK_MAKE_WARN_UNDEFINED_VARIABLES*](#)
- [*CONFIG_SDK_TOOLCHAIN_SUPPORTS_TIME_WIDE_64_BITS*](#)

CONFIG_SDK_TOOLPREFIX

Compiler toolchain path/prefix

Found in: SDK tool configuration

The prefix/path that is used to call the toolchain. The default setting assumes a crosstool-ng gcc setup that is in your PATH.

CONFIG_SDK_PYTHON

Python interpreter

Found in: SDK tool configuration

The executable name/path that is used to run python.

(Note: This option is used with the legacy GNU Make build system only.)

CONFIG_SDK_MAKE_WARN_UNDEFINED_VARIABLES

'make' warns on undefined variables

Found in: SDK tool configuration

Adds `-warn-undefined-variables` to `MAKEFLAGS`. This causes make to print a warning any time an undefined variable is referenced.

This option helps find places where a variable reference is misspelled or otherwise missing, but it can be unwanted if you have Makefiles which depend on undefined variables expanding to an empty string.

(Note: this option is used with the legacy GNU Make build system only.)

CONFIG_SDK_TOOLCHAIN_SUPPORTS_TIME_WIDE_64_BITS

Toolchain supports time_t wide 64-bits

Found in: SDK tool configuration

Enable this option in case you have a custom toolchain which supports time_t wide 64-bits. This option checks time_t is 64-bits and disables ROM time functions to use the time functions from the toolchain instead. This option allows resolving the Y2K38 problem. See "Setup Linux Toolchain from Scratch" to build a custom toolchain which supports 64-bits time_t.

Note: ESP-IDF does not currently come with any pre-compiled toolchain that supports 64-bit wide `time_t`. This will change in a future major release, but currently 64-bit `time_t` requires a custom built toolchain.

Build type

Contains:

- `CONFIG_APP_BUILD_TYPE`

CONFIG_APP_BUILD_TYPE

Application build type

Found in: *Build type*

Select the way the application is built.

By default, the application is built as a binary file in a format compatible with the ESP32 bootloader. In addition to this application, 2nd stage bootloader is also built. Application and bootloader binaries can be written into flash and loaded/executed from there.

Another option, useful for only very small and limited applications, is to only link the `.elf` file of the application, such that it can be loaded directly into RAM over JTAG. Note that since IRAM and DRAM sizes are very limited, it is not possible to build any complex application this way. However for kinds of testing and debugging, this option may provide faster iterations, since the application does not need to be written into flash. Note that at the moment, ESP-IDF does not contain all the startup code required to initialize the CPUs and ROM memory (data/bss). Therefore it is necessary to execute a bit of ROM code prior to executing the application. A `gdbinit` file may look as follows:

```
# Connect to a running instance of OpenOCD target remote :3333 # Reset and halt the target
mon reset halt # Run to a specific point in ROM code, # where most of initialization is
complete. thb *0x40007d54 c # Load the application into RAM load # Run till app_main tb
app_main c
```

Execute this `gdbinit` file as follows:

```
xtensa-esp32-elf-gdb build/app-name.elf -x gdbinit
```

Recommended `sdkconfig.defaults` for building loadable ELF files is as follows. `CONFIG_APP_BUILD_TYPE_ELF_RAM` is required, other options help reduce application memory footprint.

```
CONFIG_APP_BUILD_TYPE_ELF_RAM=y CONFIG_VFS_SUPPORT_TERMIOS=
CONFIG_NEWLIB_NANO_FORMAT=y CONFIG_ESP32_PANIC_PRINT_HALT=y
CONFIG_ESP32_DEBUG_STUBS_ENABLE=CONFIG_ESP_ERR_TO_NAME_LOOKUP=
```

Available options:

- Default (binary application + 2nd stage bootloader) (`APP_BUILD_TYPE_APP_2NDBOOT`)
- ELF file, loadable into RAM (EXPERIMENTAL)) (`APP_BUILD_TYPE_ELF_RAM`)

Application manager

Contains:

- `CONFIG_APP_COMPILE_TIME_DATE`
- `CONFIG_APP_EXCLUDE_PROJECT_VER_VAR`
- `CONFIG_APP_EXCLUDE_PROJECT_NAME_VAR`
- `CONFIG_APP_RETRIEVE_LEN_ELF_SHA`

CONFIG_APP_COMPILE_TIME_DATE

Use time/date stamp for app

Found in: Application manager

If set, then the app will be built with the current time/date stamp. It is stored in the app description structure. If not set, time/date stamp will be excluded from app image. This can be useful for getting the same binary image files made from the same source, but at different times.

CONFIG_APP_EXCLUDE_PROJECT_VER_VAR

Exclude PROJECT_VER from firmware image

Found in: Application manager

The PROJECT_VER variable from the build system will not affect the firmware image. This value will not be contained in the esp_app_desc structure.

CONFIG_APP_EXCLUDE_PROJECT_NAME_VAR

Exclude PROJECT_NAME from firmware image

Found in: Application manager

The PROJECT_NAME variable from the build system will not affect the firmware image. This value will not be contained in the esp_app_desc structure.

CONFIG_APP_RETRIEVE_LEN_ELF_SHA

The length of APP ELF SHA is stored in RAM(chars)

Found in: Application manager

At startup, the app will read this many hex characters from the embedded APP ELF SHA-256 hash value and store it in static RAM. This ensures the app ELF SHA-256 value is always available if it needs to be printed by the panic handler code. Changing this value will change the size of a static buffer, in bytes.

Bootloader config

Contains:

- [*CONFIG_BOOTLOADER_COMPILER_OPTIMIZATION*](#)
- [*CONFIG_BOOTLOADER_LOG_LEVEL*](#)
- [*CONFIG_BOOTLOADER_SPI_CUSTOM_WP_PIN*](#)
- [*CONFIG_BOOTLOADER_SPI_WP_PIN*](#)
- [*CONFIG_BOOTLOADER_VDDSDIO_BOOST*](#)
- [*CONFIG_BOOTLOADER_FACTORY_RESET*](#)
- [*CONFIG_BOOTLOADER_APP_TEST*](#)
- [*CONFIG_BOOTLOADER_HOLD_TIME_GPIO*](#)
- [*CONFIG_BOOTLOADER_WDT_ENABLE*](#)
- [*CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE*](#)
- [*CONFIG_BOOTLOADER_SKIP_VALIDATE_IN_DEEP_SLEEP*](#)
- [*CONFIG_BOOTLOADER_CUSTOM_RESERVE_RTC*](#)

CONFIG_BOOTLOADER_COMPILER_OPTIMIZATION

Bootloader optimization Level

Found in: Bootloader config

This option sets compiler optimization level (gcc -O argument) for the bootloader.

- The default “Size” setting will add the -Os flag to CFLAGS.
- The “Debug” setting will add the -Og flag to CFLAGS.
- The “Performance” setting will add the -O2 flag to CFLAGS.
- The “None” setting will add the -O0 flag to CFLAGS.

Note that custom optimization levels may be unsupported.

Available options:

- Size (-Os) (BOOTLOADER_COMPILER_OPTIMIZATION_SIZE)
- Debug (-Og) (BOOTLOADER_COMPILER_OPTIMIZATION_DEBUG)
- Optimize for performance (-O2) (BOOTLOADER_COMPILER_OPTIMIZATION_PERF)
- Debug without optimization (-O0) (BOOTLOADER_COMPILER_OPTIMIZATION_NONE)

CONFIG_BOOTLOADER_LOG_LEVEL

Bootloader log verbosity

Found in: [Bootloader config](#)

Specify how much output to see in bootloader logs.

Available options:

- No output (BOOTLOADER_LOG_LEVEL_NONE)
- Error (BOOTLOADER_LOG_LEVEL_ERROR)
- Warning (BOOTLOADER_LOG_LEVEL_WARN)
- Info (BOOTLOADER_LOG_LEVEL_INFO)
- Debug (BOOTLOADER_LOG_LEVEL_DEBUG)
- Verbose (BOOTLOADER_LOG_LEVEL_VERBOSE)

CONFIG_BOOTLOADER_SPI_CUSTOM_WP_PIN

Use custom SPI Flash WP Pin when flash pins set in eFuse (read help)

Found in: [Bootloader config](#)

This setting is only used if the SPI flash pins have been overridden by setting the eFuses SPI_PAD_CONFIG_XXX, and the SPI flash mode is QIO or QOUT.

When this is the case, the eFuse config only defines 3 of the 4 Quad I/O data pins. The WP pin (aka ESP32 pin “SD_DATA_3” or SPI flash pin “IO2”) is not specified in eFuse. The same pin is also used for external SPIRAM if it is enabled.

If this config item is set to N (default), the correct WP pin will be automatically used for any Espressif chip or module with integrated flash. If a custom setting is needed, set this config item to Y and specify the GPIO number connected to the WP.

CONFIG_BOOTLOADER_SPI_WP_PIN

Custom SPI Flash WP Pin

Found in: [Bootloader config](#)

The option “Use custom SPI Flash WP Pin” must be set or this value is ignored

If burning a customized set of SPI flash pins in eFuse and using QIO or QOUT mode for flash, set this value to the GPIO number of the SPI flash WP pin.

CONFIG_BOOTLOADER_VDDSDIO_BOOST

VDDSDIO LDO voltage

Found in: [Bootloader config](#)

If this option is enabled, and VDDSDIO LDO is set to 1.8V (using eFuse or MTDI bootstrapping pin), bootloader will change LDO settings to output 1.9V instead. This helps prevent flash chip from browning out during flash programming operations.

This option has no effect if VDDSDIO is set to 3.3V, or if the internal VDDSDIO regulator is disabled via eFuse.

Available options:

- 1.8V (BOOTLOADER_VDDSDIO_BOOST_1_8V)
- 1.9V (BOOTLOADER_VDDSDIO_BOOST_1_9V)

CONFIG_BOOTLOADER_FACTORY_RESET

GPIO triggers factory reset

Found in: [Bootloader config](#)

Allows to reset the device to factory settings: - clear one or more data partitions; - boot from “factory” partition. The factory reset will occur if there is a GPIO input pulled low while device starts up. See settings below.

CONFIG_BOOTLOADER_NUM_PIN_FACTORY_RESET

Number of the GPIO input for factory reset

Found in: [Bootloader config](#) > [CONFIG_BOOTLOADER_FACTORY_RESET](#)

The selected GPIO will be configured as an input with internal pull-up enabled. To trigger a factory reset, this GPIO must be pulled low on reset. Note that GPIO34-39 do not have an internal pullup and an external one must be provided.

CONFIG_BOOTLOADER_OTA_DATA_ERASE

Clear OTA data on factory reset (select factory partition)

Found in: [Bootloader config](#) > [CONFIG_BOOTLOADER_FACTORY_RESET](#)

The device will boot from “factory” partition (or OTA slot 0 if no factory partition is present) after a factory reset.

CONFIG_BOOTLOADER_DATA_FACTORY_RESET

Comma-separated names of partitions to clear on factory reset

Found in: [Bootloader config](#) > [CONFIG_BOOTLOADER_FACTORY_RESET](#)

Allows customers to select which data partitions will be erased while factory reset.

Specify the names of partitions as a comma-delimited with optional spaces for readability. (Like this: “nvs, phy_init, ...”) Make sure that the name specified in the partition table and here are the same. Partitions of type “app” cannot be specified here.

CONFIG_BOOTLOADER_APP_TEST

GPIO triggers boot from test app partition

Found in: [Bootloader config](#)

Allows to run the test app from “TEST” partition. A boot from “test” partition will occur if there is a GPIO input pulled low while device starts up. See settings below.

CONFIG_BOOTLOADER_NUM_PIN_APP_TEST

Number of the GPIO input to boot TEST partition

Found in: [Bootloader config](#) > [CONFIG_BOOTLOADER_APP_TEST](#)

The selected GPIO will be configured as an input with internal pull-up enabled. To trigger a test app, this GPIO must be pulled low on reset. After the GPIO input is deactivated and the device reboots, the old application will boot. (factory or OTA[x]). Note that GPIO34-39 do not have an internal pullup and an external one must be provided.

CONFIG_BOOTLOADER_HOLD_TIME_GPIO

Hold time of GPIO for reset/test mode (seconds)

Found in: [Bootloader config](#)

The GPIO must be held low continuously for this period of time after reset before a factory reset or test partition boot (as applicable) is performed.

CONFIG_BOOTLOADER_WDT_ENABLE

Use RTC watchdog in start code

Found in: [Bootloader config](#)

Tracks the execution time of startup code. If the execution time is exceeded, the RTC_WDT will restart system. It is also useful to prevent a lock up in start code caused by an unstable power source. NOTE: Tracks the execution time starts from the bootloader code - re-set timeout, while selecting the source for slow_clk - and ends calling app_main. Re-set timeout is needed due to WDT uses a SLOW_CLK clock source. After changing a frequency slow_clk a time of WDT needs to re-set for new frequency. slow_clk depends on ESP32_RTC_CLK_SRC (INTERNAL_RC or EXTERNAL_CRYSTAL).

CONFIG_BOOTLOADER_WDT_DISABLE_IN_USER_CODE

Allows RTC watchdog disable in user code

Found in: [Bootloader config](#) > [CONFIG_BOOTLOADER_WDT_ENABLE](#)

If it is set, the client must itself reset or disable rtc_wdt in their code (app_main()). Otherwise rtc_wdt will be disabled before calling app_main function. Use function rtc_wdt_feed() for resetting counter of rtc_wdt. Use function rtc_wdt_disable() for disabling rtc_wdt.

CONFIG_BOOTLOADER_WDT_TIME_MS

Timeout for RTC watchdog (ms)

Found in: [Bootloader config](#) > [CONFIG_BOOTLOADER_WDT_ENABLE](#)

Verify that this parameter is correct and more then the execution time. Pay attention to options such as reset to factory, trigger test partition and encryption on boot - these options can increase the execution time. Note: RTC_WDT will reset while encryption operations will be performed.

CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE

Enable app rollback support

Found in: [Bootloader config](#)

After updating the app, the bootloader runs a new app with the “ESP_OTA_IMG_PENDING_VERIFY” state set. This state prevents the re-run of this app. After the first boot of the new app in the user code, the function should be called to confirm the operability of the app or vice versa about its non-operability. If the app is working, then it is marked as valid. Otherwise, it is marked as not valid and rolls back to the previous working app. A reboot is

performed, and the app is booted before the software update. Note: If during the first boot a new app the power goes out or the WDT works, then roll back will happen. Rollback is possible only between the apps with the same security versions.

CONFIG_BOOTLOADER_APP_ANTI_ROLLBACK

Enable app anti-rollback support

Found in: [Bootloader config](#) > [CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE](#)

This option prevents rollback to previous firmware/application image with lower security version.

CONFIG_BOOTLOADER_APP_SECURE_VERSION

eFuse secure version of app

Found in: [Bootloader config](#) > [CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE](#) > [CONFIG_BOOTLOADER_APP_ANTI_ROLLBACK](#)

The secure version is the sequence number stored in the header of each firmware. The security version is set in the bootloader, version is recorded in the eFuse field as the number of set ones. The allocated number of bits in the efuse field for storing the security version is limited (see `BOOTLOADER_APP_SEC_VER_SIZE_EFUSE_FIELD` option).

Bootloader: When bootloader selects an app to boot, an app is selected that has a security version greater or equal that recorded in eFuse field. The app is booted with a higher (or equal) secure version.

The security version is worth increasing if in previous versions there is a significant vulnerability and their use is not acceptable.

Your partition table should has a scheme with `ota_0 + ota_1` (without factory).

CONFIG_BOOTLOADER_APP_SEC_VER_SIZE_EFUSE_FIELD

Size of the efuse secure version field

Found in: [Bootloader config](#) > [CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE](#) > [CONFIG_BOOTLOADER_APP_ANTI_ROLLBACK](#)

The size of the efuse secure version field. Its length is limited to 32 bits for ESP32 and 16 bits for ESP32S2BETA. This determines how many times the security version can be increased.

CONFIG_BOOTLOADER_EFUSE_SECURE_VERSION_EMULATE

Emulate operations with efuse secure version(only test)

Found in: [Bootloader config](#) > [CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE](#) > [CONFIG_BOOTLOADER_APP_ANTI_ROLLBACK](#)

This option allow emulate read/write operations with efuse secure version. It allow to test anti-rollback implementation without permanent write eFuse bits. In partition table should be exist this partition `emul_efuse, data, 5, , 0x2000`.

CONFIG_BOOTLOADER_SKIP_VALIDATE_IN_DEEP_SLEEP

Skip image validation when exiting deep sleep

Found in: [Bootloader config](#)

This option disables the normal validation of an image coming out of deep sleep (checksums, SHA256, and signature). This is a trade-off between wakeup performance from deep sleep, and image integrity checks.

Only enable this if you know what you are doing. It should not be used in conjunction with using `deep_sleep()` entry and changing the active OTA partition as this would skip the validation upon first load of the new OTA partition.

CONFIG_BOOTLOADER_CUSTOM_RESERVE_RTC

Reserve RTC FAST memory for custom purposes

Found in: [Bootloader config](#)

This option allows the customer to place data in the RTC FAST memory, this area remains valid when rebooted, except for power loss. This memory is located at a fixed address and is available for both the bootloader and the application. (The application and bootloader must be compiled with the same option). The RTC FAST memory has access only through `PRO_CPU`.

CONFIG_BOOTLOADER_CUSTOM_RESERVE_RTC_SIZE

Size in bytes for custom purposes

Found in: [Bootloader config](#) > [CONFIG_BOOTLOADER_CUSTOM_RESERVE_RTC](#)

This option reserves in RTC FAST memory the area for custom purposes. If you want to create your own bootloader and save more information in this area of memory, you can increase it. It must be a multiple of 4 bytes. This area (`rtc_retain_mem_t`) is reserved and has access from the bootloader and an application.

Security features

Contains:

- [CONFIG_SECURE_SIGNED_APPS_NO_SECURE_BOOT](#)
- [CONFIG_SECURE_SIGNED_APPS_SCHEME](#)
- [CONFIG_SECURE_SIGNED_ON_BOOT_NO_SECURE_BOOT](#)
- [CONFIG_SECURE_SIGNED_ON_UPDATE_NO_SECURE_BOOT](#)
- [CONFIG_SECURE_BOOT](#)
- [CONFIG_SECURE_BOOTLOADER_MODE](#)
- [CONFIG_SECURE_BOOT_BUILD_SIGNED_BINARIES](#)
- [CONFIG_SECURE_BOOT_VERIFICATION_KEY](#)
- [CONFIG_SECURE_BOOTLOADER_KEY_ENCODING](#)
- [CONFIG_SECURE_BOOT_INSECURE](#)
- [CONFIG_SECURE_FLASH_ENC_ENABLED](#)
- *Potentially insecure options*
- [CONFIG_SECURE_DISABLE_ROM_DL_MODE](#)

CONFIG_SECURE_SIGNED_APPS_NO_SECURE_BOOT

Require signed app images

Found in: [Security features](#)

Require apps to be signed to verify their integrity.

This option uses the same app signature scheme as hardware secure boot, but unlike hardware secure boot it does not prevent the bootloader from being physically updated. This means that the device can be secured against remote network access, but not physical access. Compared to using hardware Secure Boot this option is much simpler to implement.

CONFIG_SECURE_SIGNED_APPS_SCHEME

App Signing Scheme

Found in: Security features

Select the Secure App signing scheme. Depends on the Chip Revision. There are two options: 1. ECDSA based secure boot scheme. (Only choice for Secure Boot V1) Supported in ESP32 and ESP32-ECO3. 2. The RSA based secure boot scheme. (Only choice for Secure Boot V2) Supported in ESP32-ECO3. (ESP32 Chip Revision 3 onwards)

Available options:

- ECDSA (SECURE_SIGNED_APPS_ECDSA_SCHEME)
Embeds the ECDSA public key in the bootloader and signs the application with an ECDSA key.
Refer to the documentation before enabling.
- RSA (SECURE_SIGNED_APPS_RSA_SCHEME)
Appends the RSA-3072 based Signature block to the application. Refer to <Secure Boot Version 2 documentation link> before enabling.

CONFIG_SECURE_SIGNED_ON_BOOT_NO_SECURE_BOOT

Bootloader verifies app signatures

Found in: Security features

If this option is set, the bootloader will be compiled with code to verify that an app is signed before booting it.

If hardware secure boot is enabled, this option is always enabled and cannot be disabled. If hardware secure boot is not enabled, this option doesn't add significant security by itself so most users will want to leave it disabled.

CONFIG_SECURE_SIGNED_ON_UPDATE_NO_SECURE_BOOT

Verify app signature on update

Found in: Security features

If this option is set, any OTA updated apps will have the signature verified before being considered valid.

When enabled, the signature is automatically checked whenever the esp_ota_ops.h APIs are used for OTA updates, or esp_image_format.h APIs are used to verify apps.

If hardware secure boot is enabled, this option is always enabled and cannot be disabled. If hardware secure boot is not enabled, this option still adds significant security against network-based attackers by preventing spoofing of OTA updates.

CONFIG_SECURE_BOOT

Enable hardware Secure Boot in bootloader (READ DOCS FIRST)

Found in: Security features

Build a bootloader which enables Secure Boot on first boot.

Once enabled, Secure Boot will not boot a modified bootloader. The bootloader will only load a partition table or boot an app if the data has a verified digital signature. There are implications for reflashing updated apps once secure boot is enabled.

When enabling secure boot, JTAG and ROM BASIC Interpreter are permanently disabled by default.

CONFIG_SECURE_BOOT_VERSION

Select secure boot version

Found in: [Security features](#) > [CONFIG_SECURE_BOOT](#)

Select the Secure Boot Version. Depends on the Chip Revision. Secure Boot V2 is the new RSA based secure boot scheme. Supported in ESP32-ECO3. (ESP32 Chip Revision 3 onwards) Secure Boot V1 is the AES based secure boot scheme. Supported in ESP32 and ESP32-ECO3.

Available options:

- Enable Secure Boot version 1 (SECURE_BOOT_V1_ENABLED)
Build a bootloader which enables secure boot version 1 on first boot. Refer to the Secure Boot section of the ESP-IDF Programmer's Guide for this version before enabling.
- Enable Secure Boot version 2 (SECURE_BOOT_V2_ENABLED)
Build a bootloader which enables Secure Boot version 2 on first boot. Refer to Secure Boot V2 section of the ESP-IDF Programmer's Guide for this version before enabling.

CONFIG_SECURE_BOOTLOADER_MODE

Secure bootloader mode

Found in: [Security features](#)

Available options:

- One-time flash (SECURE_BOOTLOADER_ONE_TIME_FLASH)
On first boot, the bootloader will generate a key which is not readable externally or by software. A digest is generated from the bootloader image itself. This digest will be verified on each subsequent boot.
Enabling this option means that the bootloader cannot be changed after the first time it is booted.
- Reflashable (SECURE_BOOTLOADER_REFLASHABLE)
Generate a reusable secure bootloader key, derived (via SHA-256) from the secure boot signing key.
This allows the secure bootloader to be re-flashed by anyone with access to the secure boot signing key.
This option is less secure than one-time flash, because a leak of the digest key from one device allows reflashing of any device that uses it.

CONFIG_SECURE_BOOT_BUILD_SIGNED_BINARIES

Sign binaries during build

Found in: [Security features](#)

Once secure boot or signed app requirement is enabled, app images are required to be signed.

If enabled (default), these binary files are signed as part of the build process. The file named in “Secure boot private signing key” will be used to sign the image.

If disabled, unsigned app/partition data will be built. They must be signed manually using espsecure.py. Version 1 to enable ECDSA Based Secure Boot and Version 2 to enable RSA based Secure Boot. (for example, on a remote signing server.)

CONFIG_SECURE_BOOT_SIGNING_KEY

Secure boot private signing key

Found in: [Security features](#) > [CONFIG_SECURE_BOOT_BUILD_SIGNED_BINARIES](#)

Path to the key file used to sign app images.

Key file is an ECDSA private key (NIST256p curve) in PEM format for Secure Boot V1. Key file is an RSA private key in PEM format for Secure Boot V2.

Path is evaluated relative to the project directory.

You can generate a new signing key by running the following command: `espsecure.py generate_signing_key secure_boot_signing_key.pem`

See the Secure Boot section of the ESP-IDF Programmer's Guide for this version for details.

CONFIG_SECURE_BOOT_VERIFICATION_KEY

Secure boot public signature verification key

Found in: Security features

Path to a public key file used to verify signed images. Secure Boot V1: This ECDSA public key is compiled into the bootloader and/or app, to verify app images. Secure Boot V2: This RSA public key is compiled into the signature block at the end of the bootloader/app.

Key file is in raw binary format, and can be extracted from a PEM formatted private key using the `espsecure.py extract_public_key` command.

Refer to the Secure Boot section of the ESP-IDF Programmer's Guide for this version before enabling.

CONFIG_SECURE_BOOTLOADER_KEY_ENCODING

Hardware Key Encoding

Found in: Security features

In reflashable secure bootloader mode, a hardware key is derived from the signing key (with SHA-256) and can be written to eFuse with `espefuse.py`.

Normally this is a 256-bit key, but if 3/4 Coding Scheme is used on the device then the eFuse key is truncated to 192 bits.

This configuration item doesn't change any firmware code, it only changes the size of key binary which is generated at build time.

Available options:

- No encoding (256 bit key) (`SECURE_BOOTLOADER_KEY_ENCODING_256BIT`)
- 3/4 encoding (192 bit key) (`SECURE_BOOTLOADER_KEY_ENCODING_192BIT`)

CONFIG_SECURE_BOOT_INSECURE

Allow potentially insecure options

Found in: Security features

You can disable some of the default protections offered by secure boot, in order to enable testing or a custom combination of security features.

Only enable these options if you are very sure.

Refer to the Secure Boot section of the ESP-IDF Programmer's Guide for this version before enabling.

CONFIG_SECURE_FLASH_ENC_ENABLED

Enable flash encryption on boot (READ DOCS FIRST)

Found in: Security features

If this option is set, flash contents will be encrypted by the bootloader on first boot.

Note: After first boot, the system will be permanently encrypted. Re-flashing an encrypted system is complicated and not always possible.

Read *Flash Encryption* before enabling.

CONFIG_SECURE_FLASH_ENCRYPTION_KEYSIZE

Size of generated AES-XTS key

Found in: Security features > CONFIG_SECURE_FLASH_ENC_ENABLED

Size of generated AES-XTS key.

AES-128 uses a 256-bit key (32 bytes) which occupies one Efuse key block. AES-256 uses a 512-bit key (64 bytes) which occupies two Efuse key blocks.

This setting is ignored if either type of key is already burned to Efuse before the first boot. In this case, the pre-burned key is used and no new key is generated.

Available options:

- AES-128 (256-bit key) (SECURE_FLASH_ENCRYPTION_AES128)
- AES-256 (512-bit key) (SECURE_FLASH_ENCRYPTION_AES256)

CONFIG_SECURE_FLASH_ENCRYPTION_MODE

Enable usage mode

Found in: Security features > CONFIG_SECURE_FLASH_ENC_ENABLED

By default Development mode is enabled which allows UART bootloader to perform flash encryption operations

Select Release mode only for production or manufacturing. Once enabled you can not reflash using UART bootloader

Refer to the Secure Boot section of the ESP-IDF Programmer's Guide for this version and [Flash Encryption](#) for details.

Available options:

- Development(NOT SECURE) (SECURE_FLASH_ENCRYPTION_MODE_DEVELOPMENT)
- Release (SECURE_FLASH_ENCRYPTION_MODE_RELEASE)

Potentially insecure options

 Contains:

- [CONFIG_SECURE_BOOT_ALLOW_ROM_BASIC](#)
- [CONFIG_SECURE_BOOT_ALLOW_JTAG](#)
- [CONFIG_SECURE_BOOT_ALLOW_SHORT_APP_PARTITION](#)
- [CONFIG_SECURE_BOOT_V2_ALLOW_EFUSE_RD_DIS](#)
- [CONFIG_SECURE_FLASH_UART_BOOTLOADER_ALLOW_ENC](#)
- [CONFIG_SECURE_FLASH_UART_BOOTLOADER_ALLOW_DEC](#)
- [CONFIG_SECURE_FLASH_UART_BOOTLOADER_ALLOW_CACHE](#)
- [CONFIG_SECURE_FLASH_REQUIRE_ALREADY_ENABLED](#)

CONFIG_SECURE_BOOT_ALLOW_ROM_BASIC

Leave ROM BASIC Interpreter available on reset

Found in: Security features > Potentially insecure options

By default, the BASIC ROM Console starts on reset if no valid bootloader is read from the flash.

When either flash encryption or secure boot are enabled, the default is to disable this BASIC fallback mode permanently via eFuse.

If this option is set, this eFuse is not burned and the BASIC ROM Console may remain accessible. Only set this option in testing environments.

CONFIG_SECURE_BOOT_ALLOW_JTAG

Allow JTAG Debugging

Found in: Security features > Potentially insecure options

If not set (default), the bootloader will permanently disable JTAG (across entire chip) on first boot when either secure boot or flash encryption is enabled.

Setting this option leaves JTAG on for debugging, which negates all protections of flash encryption and some of the protections of secure boot.

Only set this option in testing environments.

CONFIG_SECURE_BOOT_ALLOW_SHORT_APP_PARTITION

Allow app partition length not 64KB aligned

Found in: Security features > Potentially insecure options

If not set (default), app partition size must be a multiple of 64KB. App images are padded to 64KB length, and the bootloader checks any trailing bytes after the signature (before the next 64KB boundary) have not been written. This is because flash cache maps entire 64KB pages into the address space. This prevents an attacker from appending unverified data after the app image in the flash, causing it to be mapped into the address space.

Setting this option allows the app partition length to be unaligned, and disables padding of the app image to this length. It is generally not recommended to set this option, unless you have a legacy partitioning scheme which doesn't support 64KB aligned partition lengths.

CONFIG_SECURE_BOOT_V2_ALLOW_EFUSE_RD_DIS

Allow additional read protecting of efuses

Found in: Security features > Potentially insecure options

If not set (default, recommended), on first boot the bootloader will burn the WR_DIS_RD_DIS efuse when Secure Boot is enabled. This prevents any more efuses from being read protected.

If this option is set, it will remain possible to write the EFUSE_RD_DIS efuse field after Secure Boot is enabled. This may allow an attacker to read-protect the BLK2 efuse holding the public key digest, causing an immediate denial of service and possibly allowing an additional fault injection attack to bypass the signature protection.

CONFIG_SECURE_FLASH_UART_BOOTLOADER_ALLOW_ENC

Leave UART bootloader encryption enabled

Found in: Security features > Potentially insecure options

If not set (default), the bootloader will permanently disable UART bootloader encryption access on first boot. If set, the UART bootloader will still be able to access hardware encryption.

It is recommended to only set this option in testing environments.

CONFIG_SECURE_FLASH_UART_BOOTLOADER_ALLOW_DEC

Leave UART bootloader decryption enabled

Found in: Security features > Potentially insecure options

If not set (default), the bootloader will permanently disable UART bootloader decryption access on first boot. If set, the UART bootloader will still be able to access hardware decryption.

Only set this option in testing environments. Setting this option allows complete bypass of flash encryption.

CONFIG_SECURE_FLASH_UART_BOOTLOADER_ALLOW_CACHE

Leave UART bootloader flash cache enabled

Found in: Security features > Potentially insecure options

If not set (default), the bootloader will permanently disable UART bootloader flash cache access on first boot. If set, the UART bootloader will still be able to access the flash cache.

Only set this option in testing environments.

CONFIG_SECURE_FLASH_REQUIRE_ALREADY_ENABLED

Require flash encryption to be already enabled

Found in: Security features > Potentially insecure options

If not set (default), and flash encryption is not yet enabled in eFuses, the 2nd stage bootloader will enable flash encryption: generate the flash encryption key and program eFuses. If this option is set, and flash encryption is not yet enabled, the bootloader will error out and reboot. If flash encryption is enabled in eFuses, this option does not change the bootloader behavior.

Only use this option in testing environments, to avoid accidentally enabling flash encryption on the wrong device. The device needs to have flash encryption already enabled using `espefuse.py`.

CONFIG_SECURE_DISABLE_ROM_DL_MODE

Permanently disable ROM Download Mode

Found in: Security features

If set, during startup the app will burn an eFuse bit to permanently disable the UART ROM Download Mode. This prevents any future use of `esptool.py`, `espefuse.py` and similar tools.

Once disabled, if the SoC is booted with strapping pins set for ROM Download Mode then an error is printed instead.

It is recommended to enable this option in any production application where Flash Encryption and/or Secure Boot is enabled and access to Download Mode is not required.

It is also possible to permanently disable Download Mode by calling `esp_efuse_disable_rom_download_mode()` at runtime.

Serial flasher config

Contains:

- `CONFIG_ESPTOOLPY_PORT`
- `CONFIG_ESPTOOLPY_BAUD`
- `CONFIG_ESPTOOLPY_BAUD_OTHER_VAL`
- `CONFIG_ESPTOOLPY_COMPRESSED`
- `CONFIG_ESPTOOLPY_FLASHMODE`
- `CONFIG_ESPTOOLPY_FLASHFREQ`
- `CONFIG_ESPTOOLPY_FLASHSIZE`
- `CONFIG_ESPTOOLPY_FLASHSIZE_DETECT`
- `CONFIG_ESPTOOLPY_BEFORE`
- `CONFIG_ESPTOOLPY_AFTER`
- `CONFIG_ESPTOOLPY_MONITOR_BAUD`
- `CONFIG_ESPTOOLPY_MONITOR_BAUD_OTHER_VAL`

CONFIG_ESPTOOLPY_PORT

Default serial port

Found in: Serial flasher config

The serial port that's connected to the ESP chip. This can be overridden by setting the ESPPORT environment variable.

This value is ignored when using the CMake-based build system or idf.py.

CONFIG_ESPTOOLPY_BAUD

Default baud rate

Found in: Serial flasher config

Default baud rate to use while communicating with the ESP chip. Can be overridden by setting the ESPBAUD variable.

This value is ignored when using the CMake-based build system or idf.py.

Available options:

- 115200 baud (ESPTOOLPY_BAUD_115200B)
- 230400 baud (ESPTOOLPY_BAUD_230400B)
- 921600 baud (ESPTOOLPY_BAUD_921600B)
- 2Mbaud (ESPTOOLPY_BAUD_2MB)
- Other baud rate (ESPTOOLPY_BAUD_OTHER)

CONFIG_ESPTOOLPY_BAUD_OTHER_VAL

Other baud rate value

Found in: Serial flasher config

CONFIG_ESPTOOLPY_COMPRESSED

Use compressed upload

Found in: Serial flasher config

The flasher tool can send data compressed using zlib, letting the ROM on the ESP chip decompress it on the fly before flashing it. For most payloads, this should result in a speed increase.

CONFIG_ESPTOOLPY_FLASHMODE

Flash SPI mode

Found in: Serial flasher config

Mode the flash chip is flashed in, as well as the default mode for the binary to run in.

Available options:

- QIO (ESPTOOLPY_FLASHMODE_QIO)
- QOUT (ESPTOOLPY_FLASHMODE_QOUT)
- DIO (ESPTOOLPY_FLASHMODE_DIO)
- DOUT (ESPTOOLPY_FLASHMODE_DOUT)

CONFIG_ESPTOOLPY_FLASHFREQ

Flash SPI speed

Found in: Serial flasher config

The SPI flash frequency to be used.

Available options:

- 80 MHz (ESPTOOLPY_FLASHFREQ_80M)
- 40 MHz (ESPTOOLPY_FLASHFREQ_40M)
- 26 MHz (ESPTOOLPY_FLASHFREQ_26M)
- 20 MHz (ESPTOOLPY_FLASHFREQ_20M)

CONFIG_ESPTOOLPY_FLASHSIZE

Flash size

Found in: [Serial flasher config](#)

SPI flash size, in megabytes

Available options:

- 1 MB (ESPTOOLPY_FLASHSIZE_1MB)
- 2 MB (ESPTOOLPY_FLASHSIZE_2MB)
- 4 MB (ESPTOOLPY_FLASHSIZE_4MB)
- 8 MB (ESPTOOLPY_FLASHSIZE_8MB)
- 16 MB (ESPTOOLPY_FLASHSIZE_16MB)

CONFIG_ESPTOOLPY_FLASHSIZE_DETECT

Detect flash size when flashing bootloader

Found in: [Serial flasher config](#)

If this option is set, flashing the project will automatically detect the flash size of the target chip and update the bootloader image before it is flashed.

CONFIG_ESPTOOLPY_BEFORE

Before flashing

Found in: [Serial flasher config](#)

Configure whether esptool.py should reset the ESP32 before flashing.

Automatic resetting depends on the RTS & DTR signals being wired from the serial port to the ESP32. Most USB development boards do this internally.

Available options:

- Reset to bootloader (ESPTOOLPY_BEFORE_RESET)
- No reset (ESPTOOLPY_BEFORE_NORESET)

CONFIG_ESPTOOLPY_AFTER

After flashing

Found in: [Serial flasher config](#)

Configure whether esptool.py should reset the ESP32 after flashing.

Automatic resetting depends on the RTS & DTR signals being wired from the serial port to the ESP32. Most USB development boards do this internally.

Available options:

- Reset after flashing (ESPTOOLPY_AFTER_RESET)
- Stay in bootloader (ESPTOOLPY_AFTER_NORESET)

CONFIG_ESPTOOLPY_MONITOR_BAUD

‘idf.py monitor’ baud rate

Found in: Serial flasher config

Baud rate to use when running ‘idf.py monitor’ or ‘make monitor’ to view serial output from a running chip.

Can override by setting the MONITORBAUD environment variable.

Available options:

- 9600 bps (ESPTOOLPY_MONITOR_BAUD_9600B)
- 57600 bps (ESPTOOLPY_MONITOR_BAUD_57600B)
- 115200 bps (ESPTOOLPY_MONITOR_BAUD_115200B)
- 230400 bps (ESPTOOLPY_MONITOR_BAUD_230400B)
- 921600 bps (ESPTOOLPY_MONITOR_BAUD_921600B)
- 2 Mbps (ESPTOOLPY_MONITOR_BAUD_2MB)
- Custom baud rate (ESPTOOLPY_MONITOR_BAUD_OTHER)

CONFIG_ESPTOOLPY_MONITOR_BAUD_OTHER_VAL

Custom baud rate value

Found in: Serial flasher config

Partition Table

Contains:

- *CONFIG_PARTITION_TABLE_TYPE*
- *CONFIG_PARTITION_TABLE_CUSTOM_FILENAME*
- *CONFIG_PARTITION_TABLE_OFFSET*
- *CONFIG_PARTITION_TABLE_MD5*

CONFIG_PARTITION_TABLE_TYPE

Partition Table

Found in: Partition Table

The partition table to flash to the ESP32. The partition table determines where apps, data and other resources are expected to be found.

The predefined partition table CSV descriptions can be found in the components/partition_table directory. Otherwise it’s possible to create a new custom partition CSV for your application.

Available options:

- Single factory app, no OTA (PARTITION_TABLE_SINGLE_APP)
- Factory app, two OTA definitions (PARTITION_TABLE_TWO_OTA)
- Custom partition table CSV (PARTITION_TABLE_CUSTOM)

CONFIG_PARTITION_TABLE_CUSTOM_FILENAME

Custom partition CSV file

Found in: Partition Table

Name of the custom partition CSV filename. This path is evaluated relative to the project root directory.

CONFIG_PARTITION_TABLE_OFFSET

Offset of partition table

Found in: Partition Table

The address of partition table (by default 0x8000). Allows you to move the partition table, it gives more space for the bootloader. Note that the bootloader and app will both need to be compiled with the same PARTITION_TABLE_OFFSET value.

This number should be a multiple of 0x1000.

Note that partition offsets in the partition table CSV file may need to be changed if this value is set to a higher value. To have each partition offset adapt to the configured partition table offset, leave all partition offsets blank in the CSV file.

CONFIG_PARTITION_TABLE_MD5

Generate an MD5 checksum for the partition table

Found in: Partition Table

Generate an MD5 checksum for the partition table for protecting the integrity of the table. The generation should be turned off for legacy bootloaders which cannot recognize the MD5 checksum in the partition table.

Compiler options

Contains:

- *CONFIG_COMPILER_OPTIMIZATION*
- *CONFIG_COMPILER_OPTIMIZATION_ASSERTION_LEVEL*
- *CONFIG_COMPILER_CXX_EXCEPTIONS*
- *CONFIG_COMPILER_CXX_RTTI*
- *CONFIG_COMPILER_STACK_CHECK_MODE*
- *CONFIG_COMPILER_WARN_WRITE_STRINGS*
- *CONFIG_COMPILER_DISABLE_GCC8_WARNINGS*

CONFIG_COMPILER_OPTIMIZATION

Optimization Level

Found in: Compiler options

This option sets compiler optimization level (gcc -O argument) for the app.

- The “Default” setting will add the -Og flag to CFLAGS.
- The “Size” setting will add the -Os flag to CFLAGS.
- The “Performance” setting will add the -O2 flag to CFLAGS.
- The “None” setting will add the -O0 flag to CFLAGS.

The “Size” setting cause the compiled code to be smaller and faster, but may lead to difficulties of correlating code addresses to source file lines when debugging.

The “Performance” setting causes the compiled code to be larger and faster, but will be easier to correlated code addresses to source file lines.

“None” with -O0 produces compiled code without optimization.

Note that custom optimization levels may be unsupported.

Compiler optimization for the IDF bootloader is set separately, see the BOOT-LOADER_COMPILER_OPTIMIZATION setting.

Available options:

- Debug (-Og) (COMPILER_OPTIMIZATION_DEFAULT)
- Optimize for size (-Os) (COMPILER_OPTIMIZATION_SIZE)
- Optimize for performance (-O2) (COMPILER_OPTIMIZATION_PERF)
- Debug without optimization (-O0) (COMPILER_OPTIMIZATION_NONE)

CONFIG_COMPILER_OPTIMIZATION_ASSERTION_LEVEL

Assertion level

Found in: [Compiler options](#)

Assertions can be:

- Enabled. Failure will print verbose assertion details. This is the default.
- Set to “silent” to save code size (failed assertions will abort() but user needs to use the aborting address to find the line number with the failed assertion.)
- Disabled entirely (not recommended for most configurations.) -DNDEBUG is added to CPPFLAGS in this case.

Available options:

- Enabled (COMPILER_OPTIMIZATION_ASSERTIONS_ENABLE)
Enable assertions. Assertion content and line number will be printed on failure.
- Silent (saves code size) (COMPILER_OPTIMIZATION_ASSERTIONS_SILENT)
Enable silent assertions. Failed assertions will abort(), user needs to use the aborting address to find the line number with the failed assertion.
- Disabled (sets -DNDEBUG) (COMPILER_OPTIMIZATION_ASSERTIONS_DISABLE)
If assertions are disabled, -DNDEBUG is added to CPPFLAGS.

CONFIG_COMPILER_CXX_EXCEPTIONS

Enable C++ exceptions

Found in: [Compiler options](#)

Enabling this option compiles all IDF C++ files with exception support enabled.

Disabling this option disables C++ exception support in all compiled files, and any libstdc++ code which throws an exception will abort instead.

Enabling this option currently adds an additional ~500 bytes of heap overhead when an exception is thrown in user code for the first time.

Contains:

- [CONFIG_COMPILER_CXX_EXCEPTIONS_EMG_POOL_SIZE](#)

CONFIG_COMPILER_CXX_EXCEPTIONS_EMG_POOL_SIZE

Emergency Pool Size

Found in: [Compiler options](#) > [CONFIG_COMPILER_CXX_EXCEPTIONS](#)

Size (in bytes) of the emergency memory pool for C++ exceptions. This pool will be used to allocate memory for thrown exceptions when there is not enough memory on the heap.

CONFIG_COMPILER_CXX_RTTI

Enable C++ run-time type info (RTTI)

Found in: [Compiler options](#)

Enabling this option compiles all C++ files with RTTI support enabled. This increases binary size (typically by tens of kB) but allows using dynamic_cast conversion and typeid operator.

CONFIG_COMPILER_STACK_CHECK_MODE

Stack smashing protection mode

Found in: *Compiler options*

Stack smashing protection mode. Emit extra code to check for buffer overflows, such as stack smashing attacks. This is done by adding a guard variable to functions with vulnerable objects. The guards are initialized when a function is entered and then checked when the function exits. If a guard check fails, program is halted. Protection has the following modes:

- In NORMAL mode (GCC flag: `-fstack-protector`) only functions that call `alloca`, and functions with buffers larger than 8 bytes are protected.
- STRONG mode (GCC flag: `-fstack-protector-strong`) is like NORMAL, but includes additional functions to be protected –those that have local array definitions, or have references to local frame addresses.
- In OVERALL mode (GCC flag: `-fstack-protector-all`) all functions are protected.

Modes have the following impact on code performance and coverage:

- performance: NORMAL > STRONG > OVERALL
- coverage: NORMAL < STRONG < OVERALL

Available options:

- None (COMPILER_STACK_CHECK_MODE_NONE)
- Normal (COMPILER_STACK_CHECK_MODE_NORM)
- Strong (COMPILER_STACK_CHECK_MODE_STRONG)
- Overall (COMPILER_STACK_CHECK_MODE_ALL)

CONFIG_COMPILER_WARN_WRITE_STRINGS

Enable `-Wwrite-strings` warning flag

Found in: *Compiler options*

Adds `-Wwrite-strings` flag for the C/C++ compilers.

For C, this gives string constants the type `const char []` so that copying the address of one into a non-const `char *` pointer produces a warning. This warning helps to find at compile time code that tries to write into a string constant.

For C++, this warns about the deprecated conversion from string literals to `char *`.

CONFIG_COMPILER_DISABLE_GCC8_WARNINGS

Disable new warnings introduced in GCC 6 - 8

Found in: *Compiler options*

Enable this option if using GCC 6 or newer, and wanting to disable warnings which don't appear with GCC 5.

Component config

Contains:

- *Application Level Tracing*
- *Bluetooth*
- *CONFIG_BLE_MESH*
- *CoAP Configuration*
- *Driver configurations*
- *eFuse Bit Manager*
- *ESP-TLS*
- *ESP32-specific*

- *Power Management*
- *ADC-Calibration*
- *Common ESP-related*
- *Ethernet*
- *Event Loop Library*
- *GDB Stub*
- *ESP HTTP client*
- *HTTP Server*
- *ESP HTTPS OTA*
- *ESP HTTPS server*
- *ESP NETIF Adapter*
- *Wi-Fi*
- *PHY*
- *Core dump*
- *FAT Filesystem support*
- *Modbus configuration*
- *FreeRTOS*
- *Heap memory debugging*
- *jsmn*
- *libsodium*
- *Log output*
- *LWIP*
- *mbedTLS*
- *mDNS*
- *ESP-MQTT Configurations*
- *Newlib*
- *NVS*
- *OpenSSL*
- *PThreads*
- *SPI Flash driver*
- *SPIFFS Configuration*
- *Unity unit testing library*
- *Virtual file system*
- *Wear Levelling*
- *Wi-Fi Provisioning Manager*
- *Supplicant*

Application Level Tracing Contains:

- *CONFIG_APPTRACE_DESTINATION*
- *CONFIG_APPTRACE_ONPANIC_HOST_FLUSH_TMO*
- *CONFIG_APPTRACE_POSTMORTEM_FLUSH_THRESH*
- *CONFIG_APPTRACE_PENDING_DATA_SIZE_MAX*
- *FreeRTOS System View Tracing*
- *CONFIG_APPTRACE_GCOV_ENABLE*

CONFIG_APPTRACE_DESTINATION

Data Destination

Found in: Component config > Application Level Tracing

Select destination for application trace: trace memory or none (to disable).

Available options:

- Trace memory (APPTRACE_DEST_TRAX)
- None (APPTRACE_DEST_NONE)

CONFIG_APPTRACE_ONPANIC_HOST_FLUSH_TMO

Timeout for flushing last trace data to host on panic

Found in: [Component config](#) > [Application Level Tracing](#)

Timeout for flushing last trace data to host in case of panic. In ms. Use -1 to disable timeout and wait forever.

CONFIG_APPTRACE_POSTMORTEM_FLUSH_THRESH

Threshold for flushing last trace data to host on panic

Found in: [Component config](#) > [Application Level Tracing](#)

Threshold for flushing last trace data to host on panic in post-mortem mode. This is minimal amount of data needed to perform flush. In bytes.

CONFIG_APPTRACE_PENDING_DATA_SIZE_MAX

Size of the pending data buffer

Found in: [Component config](#) > [Application Level Tracing](#)

Size of the buffer for events in bytes. It is useful for buffering events from the time critical code (scheduler, ISRs etc). If this parameter is 0 then events will be discarded when main HW buffer is full.

FreeRTOS SystemView Tracing Contains:

- [CONFIG_SYSVIEW_ENABLE](#)

CONFIG_SYSVIEW_ENABLE

SystemView Tracing Enable

Found in: [Component config](#) > [Application Level Tracing](#) > [FreeRTOS SystemView Tracing](#)

Enables support for SEGGER SystemView tracing functionality.

CONFIG_SYSVIEW_TS_SOURCE

Timer to use as timestamp source

Found in: [Component config](#) > [Application Level Tracing](#) > [FreeRTOS SystemView Tracing](#) > [CONFIG_SYSVIEW_ENABLE](#)

SystemView needs to use a hardware timer as the source of timestamps when tracing. This option selects the timer for it.

Available options:

- CPU cycle counter (CCOUNT) (SYSVIEW_TS_SOURCE_CCOUNT)
- Timer 0, Group 0 (SYSVIEW_TS_SOURCE_TIMER_00)
- Timer 1, Group 0 (SYSVIEW_TS_SOURCE_TIMER_01)
- Timer 0, Group 1 (SYSVIEW_TS_SOURCE_TIMER_10)
- Timer 1, Group 1 (SYSVIEW_TS_SOURCE_TIMER_11)
- esp_timer high resolution timer (SYSVIEW_TS_SOURCE_ESP_TIMER)

CONFIG_SYSVIEW_MAX_TASKS

Maximum supported tasks

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing > CONFIG_SYSVIEW_ENABLE

Configures maximum supported tasks in sysview debug

CONFIG_SYSVIEW_BUF_WAIT_TMO

Trace buffer wait timeout

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing > CONFIG_SYSVIEW_ENABLE

Configures timeout (in us) to wait for free space in trace buffer. Set to -1 to wait forever and avoid lost events.

CONFIG_SYSVIEW_EVT_OVERFLOW_ENABLE

Trace Buffer Overflow Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing > CONFIG_SYSVIEW_ENABLE

Enables “Trace Buffer Overflow” event.

CONFIG_SYSVIEW_EVT_ISR_ENTER_ENABLE

ISR Enter Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing > CONFIG_SYSVIEW_ENABLE

Enables “ISR Enter” event.

CONFIG_SYSVIEW_EVT_ISR_EXIT_ENABLE

ISR Exit Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing > CONFIG_SYSVIEW_ENABLE

Enables “ISR Exit” event.

CONFIG_SYSVIEW_EVT_ISR_TO_SCHEDULER_ENABLE

ISR Exit to Scheduler Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing > CONFIG_SYSVIEW_ENABLE

Enables “ISR to Scheduler” event.

CONFIG_SYSVIEW_EVT_TASK_START_EXEC_ENABLE

Task Start Execution Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing > CONFIG_SYSVIEW_ENABLE

Enables “Task Start Execution” event.

CONFIG_SYSVIEW_EVT_TASK_STOP_EXEC_ENABLE

Task Stop Execution Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing > CONFIG_SYSVIEW_ENABLE

Enables “Task Stop Execution” event.

CONFIG_SYSVIEW_EVT_TASK_START_READY_ENABLE

Task Start Ready State Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing > CONFIG_SYSVIEW_ENABLE

Enables “Task Start Ready State” event.

CONFIG_SYSVIEW_EVT_TASK_STOP_READY_ENABLE

Task Stop Ready State Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing > CONFIG_SYSVIEW_ENABLE

Enables “Task Stop Ready State” event.

CONFIG_SYSVIEW_EVT_TASK_CREATE_ENABLE

Task Create Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing > CONFIG_SYSVIEW_ENABLE

Enables “Task Create” event.

CONFIG_SYSVIEW_EVT_TASK_TERMINATE_ENABLE

Task Terminate Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing > CONFIG_SYSVIEW_ENABLE

Enables “Task Terminate” event.

CONFIG_SYSVIEW_EVT_IDLE_ENABLE

System Idle Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing > CONFIG_SYSVIEW_ENABLE

Enables “System Idle” event.

CONFIG_SYSVIEW_EVT_TIMER_ENTER_ENABLE

Timer Enter Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing > CONFIG_SYSVIEW_ENABLE

Enables “Timer Enter” event.

CONFIG_SYSVIEW_EVT_TIMER_EXIT_ENABLE

Timer Exit Event

Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing > CONFIG_SYSVIEW_ENABLE

Enables “Timer Exit” event.

CONFIG_APPTRACE_GCOV_ENABLE

GCOV to Host Enable

Found in: Component config > Application Level Tracing

Enables support for GCOV data transfer to host.

Bluetooth Contains:

- *CONFIG_BT_ENABLED*
- *Bluetooth controller*
- *CONFIG_BT_HOST*
- *Bluedroid Options*
- *NimBLE Options*

CONFIG_BT_ENABLED

Bluetooth

Found in: Component config > Bluetooth

Select this option to enable Bluetooth and show the submenu with Bluetooth configuration choices.

Bluetooth controller Contains:

- *CONFIG_BTDM_CTRL_MODE*
- *CONFIG_BTDM_CTRL_BLE_MAX_CONN*
- *CONFIG_BTDM_CTRL_BR_EDR_MAX_ACL_CONN*
- *CONFIG_BTDM_CTRL_BR_EDR_MAX_SYNC_CONN*
- *CONFIG_BTDM_CTRL_BR_EDR_SCO_DATA_PATH*
- *CONFIG_BTDM_CTRL_PCM_ROLE_EDGE_CONFIG*
- *CONFIG_BTDM_CTRL_AUTO_LATENCY*
- *CONFIG_BTDM_CTRL_LEGACY_AUTH_VENDOR_EVT*
- *CONFIG_BTDM_CTRL_PINNED_TO_CORE_CHOICE*
- *CONFIG_BTDM_CTRL_HCI_MODE_CHOICE*
- *HCI UART(H4) Options*
- *MODEM SLEEP Options*
- *CONFIG_BTDM_BLE_SLEEP_CLOCK_ACCURACY*
- *CONFIG_BTDM_BLE_SCAN_DUPL*
- *CONFIG_BTDM_CTRL_FULL_SCAN_SUPPORTED*
- *CONFIG_BTDM_BLE_ADV_REPORT_FLOW_CTRL_SUPP*

CONFIG_BTDM_CTRL_MODE

Bluetooth controller mode (BR/EDR/BLE/DUALMODE)

Found in: Component config > Bluetooth > Bluetooth controller

Specify the bluetooth controller mode (BR/EDR, BLE or dual mode).

Available options:

- BLE Only (BTDM_CTRL_MODE_BLE_ONLY)

- BR/EDR Only (BTDM_CTRL_MODE_BR_EDR_ONLY)
- Bluetooth Dual Mode (BTDM_CTRL_MODE_BTDM)

CONFIG_BTDM_CTRL_BLE_MAX_CONN

BLE Max Connections

Found in: [Component config](#) > [Bluetooth](#) > [Bluetooth controller](#)

BLE maximum connections of bluetooth controller. Each connection uses 1KB static DRAM whenever the BT controller is enabled.

CONFIG_BTDM_CTRL_BR_EDR_MAX_ACL_CONN

BR/EDR ACL Max Connections

Found in: [Component config](#) > [Bluetooth](#) > [Bluetooth controller](#)

BR/EDR ACL maximum connections of bluetooth controller. Each connection uses 1.2KB static DRAM whenever the BT controller is enabled.

CONFIG_BTDM_CTRL_BR_EDR_MAX_SYNC_CONN

BR/EDR Sync(SCO/eSCO) Max Connections

Found in: [Component config](#) > [Bluetooth](#) > [Bluetooth controller](#)

BR/EDR Synchronize maximum connections of bluetooth controller. Each connection uses 2KB static DRAM whenever the BT controller is enabled.

CONFIG_BTDM_CTRL_BR_EDR_SCO_DATA_PATH

BR/EDR Sync(SCO/eSCO) default data path

Found in: [Component config](#) > [Bluetooth](#) > [Bluetooth controller](#)

SCO data path, i.e. HCI or PCM. SCO data can be sent/received through HCI synchronous packets, or the data can be routed to on-chip PCM module on ESP32. PCM input/output signals can be “matrixed” to GPIOs. The default data path can also be set using API “esp_bredr_sco_datapath_set”

Available options:

- HCI (BTDM_CTRL_BR_EDR_SCO_DATA_PATH_HCI)
- PCM (BTDM_CTRL_BR_EDR_SCO_DATA_PATH_PCM)

CONFIG_BTDM_CTRL_PCM_ROLE_EDGE_CONFIG

PCM Signal Config (Role and Polar)

Found in: [Component config](#) > [Bluetooth](#) > [Bluetooth controller](#)

Contains:

- [CONFIG_BTDM_CTRL_PCM_ROLE](#)
- [CONFIG_BTDM_CTRL_PCM_POLAR](#)

CONFIG_BTDM_CTRL_PCM_ROLE

PCM Role

Found in: [Component config](#) > [Bluetooth](#) > [Bluetooth controller](#) > [CONFIG_BTDM_CTRL_PCM_ROLE_EDGE_CONFIG](#)

PCM role can be configured as PCM master or PCM slave

Available options:

- PCM Master (BTDM_CTRL_PCM_ROLE_MASTER)
- PCM Slave (BTDM_CTRL_PCM_ROLE_SLAVE)

CONFIG_BTDM_CTRL_PCM_POLAR

PCM Polar

Found in: Component config > Bluetooth > Bluetooth controller > CONFIG_BTDM_CTRL_PCM_ROLE_EDGE_CONFIG

PCM polarity can be configured as Falling Edge or Rising Edge

Available options:

- Falling Edge (BTDM_CTRL_PCM_POLAR_FALLING_EDGE)
- Rising Edge (BTDM_CTRL_PCM_POLAR_RISING_EDGE)

CONFIG_BTDM_CTRL_AUTO_LATENCY

Auto latency

Found in: Component config > Bluetooth > Bluetooth controller

BLE auto latency, used to enhance classic BT performance while classic BT and BLE are enabled at the same time.

CONFIG_BTDM_CTRL_LEGACY_AUTH_VENDOR_EVT

Legacy Authentication Vendor Specific Event Enable

Found in: Component config > Bluetooth > Bluetooth controller

To protect from BIAS attack during Legacy authentication, Legacy authentication Vendor specific event should be enabled

CONFIG_BTDM_CTRL_PINNED_TO_CORE_CHOICE

The cpu core which bluetooth controller run

Found in: Component config > Bluetooth > Bluetooth controller

Specify the cpu core to run bluetooth controller. Can not specify no-affinity.

Available options:

- Core 0 (PRO CPU) (BTDM_CTRL_PINNED_TO_CORE_0)
- Core 1 (APP CPU) (BTDM_CTRL_PINNED_TO_CORE_1)

CONFIG_BTDM_CTRL_HCI_MODE_CHOICE

HCI mode

Found in: Component config > Bluetooth > Bluetooth controller

Specify HCI mode as VHCI or UART(H4)

Available options:

- VHCI (BTDM_CTRL_HCI_MODE_VHCI)
Normal option. Mostly, choose this VHCI when bluetooth host run on ESP32, too.
- UART(H4) (BTDM_CTRL_HCI_MODE_UART_H4)
If use external bluetooth host which run on other hardware and use UART as the HCI interface, choose this option.

HCI UART(H4) Options Contains:

- [CONFIG_BT_HCI_UART_NO](#)
- [CONFIG_BT_HCI_UART_BAUDRATE](#)

CONFIG_BT_HCI_UART_NO

UART Number for HCI

Found in: Component config > Bluetooth > Bluetooth controller > HCI UART(H4) Options

Uart number for HCI. The available uart is UART1 and UART2.

CONFIG_BT_HCI_UART_BAUDRATE

UART Baudrate for HCI

Found in: Component config > Bluetooth > Bluetooth controller > HCI UART(H4) Options

UART Baudrate for HCI. Please use standard baudrate.

MODEM SLEEP Options Contains:

- [CONFIG_BTDM_MODEM_SLEEP](#)
- [CONFIG_BTDM_LOW_POWER_CLOCK](#)

CONFIG_BTDM_MODEM_SLEEP

Bluetooth modem sleep

Found in: Component config > Bluetooth > Bluetooth controller > MODEM SLEEP Options

Enable/disable bluetooth controller low power mode.

CONFIG_BTDM_MODEM_SLEEP_MODE

Bluetooth Modem sleep mode

Found in: Component config > Bluetooth > Bluetooth controller > MODEM SLEEP Options > CONFIG_BTDM_MODEM_SLEEP

To select which strategy to use for modem sleep

Available options:

- **ORIG Mode(sleep with low power clock) (BTDM_MODEM_SLEEP_MODE_ORIG)**
ORIG mode is a bluetooth sleep mode that can be used for dual mode controller. In this mode, bluetooth controller sleeps between BR/EDR frames and BLE events. A low power clock is used to maintain bluetooth reference clock.
- **EVED Mode(For internal test only) (BTDM_MODEM_SLEEP_MODE_EVED)**
EVED mode is for BLE only and is only for internal test. Do not use it for production. this mode is not compatible with DFS nor light sleep

CONFIG_BTDM_LOW_POWER_CLOCK

Bluetooth low power clock

Found in: Component config > Bluetooth > Bluetooth controller > MODEM SLEEP Options

Select the low power clock source for bluetooth controller. Bluetooth low power clock is the clock source to maintain time in sleep mode.

- “Main crystal” option provides good accuracy and can support Dynamic Frequency Scaling to be used with Bluetooth modem sleep. Light sleep is not supported.

- “External 32kHz crystal” option allows user to use a 32.768kHz crystal as Bluetooth low power clock. This option is allowed as long as External 32kHz crystal is configured as the system RTC clock source. This option provides good accuracy and supports Bluetooth modem sleep to be used alongside Dynamic Frequency Scaling or light sleep.

Available options:

- Main crystal (BTDM_LPCLK_SEL_MAIN_XTAL)
Main crystal can be used as low power clock for bluetooth modem sleep. If this option is selected, bluetooth modem sleep can work under Dynamic Frequency Scaling(DFS) enabled, but cannot work when light sleep is enabled. Main crystal has a good performance in accuracy as the bluetooth low power clock source.
- External 32kHz crystal (BTDM_LPCLK_SEL_EXT_32K_XTAL)
External 32kHz crystal has a nominal frequency of 32.768kHz and provides good frequency stability. If used as Bluetooth low power clock, External 32kHz can support Bluetooth modem sleep to be used with both DFS and light sleep.

CONFIG_BTDM_BLE_SLEEP_CLOCK_ACCURACY

BLE Sleep Clock Accuracy

Found in: Component config > Bluetooth > Bluetooth controller

BLE Sleep Clock Accuracy(SCA) for the local device is used to estimate window widening in BLE connection events. With a lower level of clock accuracy(e.g. 500ppm over 250ppm), the slave needs a larger RX window to synchronize with master in each anchor point, thus resulting in an increase of power consumption but a higher level of robustness in keeping connected. According to the requirements of Bluetooth Core specification 4.2, the worst-case accuracy of Classic Bluetooth low power oscillator(LPO) is +/-250ppm in STANDBY and in low power modes such as sniff. For BLE the worst-case SCA is +/-500ppm.

- “151ppm to 250ppm” option is the default value for Bluetooth Dual mode
- **“251ppm to 500ppm” option can be used in BLE only mode when using external 32kHz crystal as low power clock.** This option is provided in case that BLE sleep clock has a lower level of accuracy, or other error sources contribute to the inaccurate timing during sleep.

Available options:

- 251ppm to 500ppm (BTDM_BLE_DEFAULT_SCA_500PPM)
- 151ppm to 250ppm (BTDM_BLE_DEFAULT_SCA_250PPM)

CONFIG_BTDM_BLE_SCAN_DUPL

BLE Scan Duplicate Options

Found in: Component config > Bluetooth > Bluetooth controller

This select enables parameters setting of BLE scan duplicate.

CONFIG_BTDM_SCAN_DUPL_TYPE

Scan Duplicate Type

Found in: Component config > Bluetooth > Bluetooth controller > CONFIG_BTDM_BLE_SCAN_DUPL

Scan duplicate have three ways. one is “Scan Duplicate By Device Address”, This way is to use advertiser address filtering. The adv packet of the same address is only allowed to be reported once. Another way is “Scan Duplicate By Device Address And Advertising Data” . This way is to use advertising data and device address filtering. All different adv packets with the same address are allowed to be reported. The last way is “Scan Duplicate By Advertising Data” . This way is to use advertising data filtering. All same advertising data only allow to be reported once even though they are from different devices.

Available options:

- Scan Duplicate By Device Address (BTDM_SCAN_DUPL_TYPE_DEVICE)
This way is to use advertiser address filtering. The adv packet of the same address is only allowed to be reported once
- Scan Duplicate By Advertising Data (BTDM_SCAN_DUPL_TYPE_DATA)
This way is to use advertising data filtering. All same advertising data only allow to be reported once even though they are from different devices.
- Scan Duplicate By Device Address And Advertising Data (BTDM_SCAN_DUPL_TYPE_DATA_DEVICE)
This way is to use advertising data and device address filtering. All different adv packets with the same address are allowed to be reported.

CONFIG_BTDM_SCAN_DUPL_CACHE_SIZE

Maximum number of devices in scan duplicate filter

Found in: [Component config](#) > [Bluetooth](#) > [Bluetooth controller](#) > [CONFIG_BTDM_BLE_SCAN_DUPL](#)

Maximum number of devices which can be recorded in scan duplicate filter. When the maximum amount of device in the filter is reached, the cache will be refreshed.

CONFIG_BTDM_BLE_MESH_SCAN_DUPL_EN

Special duplicate scan mechanism for BLE Mesh scan

Found in: [Component config](#) > [Bluetooth](#) > [Bluetooth controller](#) > [CONFIG_BTDM_BLE_SCAN_DUPL](#)

This enables the BLE scan duplicate for special BLE Mesh scan.

CONFIG_BTDM_MESH_DUPL_SCAN_CACHE_SIZE

Maximum number of Mesh adv packets in scan duplicate filter

Found in: [Component config](#) > [Bluetooth](#) > [Bluetooth controller](#) > [CONFIG_BTDM_BLE_SCAN_DUPL](#) > [CONFIG_BTDM_BLE_MESH_SCAN_DUPL_EN](#)

Maximum number of adv packets which can be recorded in duplicate scan cache for BLE Mesh. When the maximum amount of device in the filter is reached, the cache will be refreshed.

CONFIG_BTDM_CTRL_FULL_SCAN_SUPPORTED

BLE full scan feature supported

Found in: [Component config](#) > [Bluetooth](#) > [Bluetooth controller](#)

The full scan function is mainly used to provide BLE scan performance. This is required for scenes with high scan performance requirements, such as BLE Mesh scenes.

CONFIG_BTDM_BLE_ADV_REPORT_FLOW_CTRL_SUPP

BLE adv report flow control supported

Found in: [Component config](#) > [Bluetooth](#) > [Bluetooth controller](#)

The function is mainly used to enable flow control for advertising reports. When it is enabled, advertising reports will be discarded by the controller if the number of unprocessed advertising reports exceeds the size of BLE adv report flow control.

CONFIG_BTDM_BLE_ADV_REPORT_FLOW_CTRL_NUM

BLE adv report flow control number

Found in: [Component config](#) > [Bluetooth](#) > [Bluetooth controller](#) > [CONFIG_BTDM_BLE_ADV_REPORT_FLOW_CTRL_SUPP](#)

The number of unprocessed advertising report that Bluebird can save. If you set `BTDM_BLE_ADV_REPORT_FLOW_CTRL_NUM` to a small value, this may cause adv packets lost. If you set `BTDM_BLE_ADV_REPORT_FLOW_CTRL_NUM` to a large value, Bluebird may cache a lot of adv packets and this may cause system memory run out. For example, if you set it to 50, the maximum memory consumed by host is 35 * 50 bytes. Please set `BTDM_BLE_ADV_REPORT_FLOW_CTRL_NUM` according to your system free memory and handle adv packets as fast as possible, otherwise it will cause adv packets lost.

CONFIG_BTDM_BLE_ADV_REPORT_DISCARD_THRSHOLD

BLE adv lost event threshold value

Found in: [Component config](#) > [Bluetooth](#) > [Bluetooth controller](#) > [CONFIG_BTDM_BLE_ADV_REPORT_FLOW_CTRL_SUPP](#)

When adv report flow control is enabled, The ADV lost event will be generated when the number of ADV packets lost in the controller reaches this threshold. It is better to set a larger value. If you set `BTDM_BLE_ADV_REPORT_DISCARD_THRSHOLD` to a small value or printf every adv lost event, it may cause adv packets lost more.

CONFIG_BT_HOST

Bluetooth Host

Found in: [Component config](#) > [Bluetooth](#)

This helps to choose Bluetooth host stack

Available options:

- Bluebird - Dual-mode (`BT_BLUEBIRD_ENABLED`)
This option is recommended for classic Bluetooth or for dual-mode usecases
- NimBLE - BLE only (`BT_NIMBLE_ENABLED`)
This option is recommended for BLE only usecases to save on memory
- Controller Only (`BT_CONTROLLER_ONLY`)
This option is recommended when you want to communicate directly with the controller (without any host) or when you are using any other host stack not supported by Espressif (not mentioned here).

Bluebird Options

 Contains:

- `CONFIG_BT_BTC_TASK_STACK_SIZE`
- `CONFIG_BT_BLUEBIRD_PINNED_TO_CORE_CHOICE`
- `CONFIG_BT_BTU_TASK_STACK_SIZE`
- `CONFIG_BT_BLUEBIRD_MEM_DEBUG`
- `CONFIG_BT_CLASSIC_ENABLED`
- `CONFIG_BT_HFP_WBS_ENABLE`
- `CONFIG_BT_SSP_ENABLED`
- `CONFIG_BT_BLE_ENABLED`
- `CONFIG_BT_STACK_NO_LOG`
- `BT_DEBUG LOG LEVEL`
- `CONFIG_BT_ACL_CONNECTIONS`
- `CONFIG_BT_ALLOCATION_FROM_SPIRAM_FIRST`
- `CONFIG_BT_BLE_DYNAMIC_ENV_MEMORY`
- `CONFIG_BT_BLE_HOST_QUEUE_CONG_CHECK`

- [CONFIG_BT_BLE_ACT_SCAN_REP_ADV_SCAN](#)
- [CONFIG_BT_BLE_ESTAB_LINK_CONN_TOUT](#)

CONFIG_BT_BTC_TASK_STACK_SIZE

Bluetooth event (callback to application) task stack size

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#)

This select btc task stack size

CONFIG_BT_BLUEDROID_PINNED_TO_CORE_CHOICE

The cpu core which Bluedroid run

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#)

Which the cpu core to run Bluedroid. Can choose core0 and core1. Can not specify no-affinity.

Available options:

- Core 0 (PRO CPU) ([BT_BLUEDROID_PINNED_TO_CORE_0](#))
- Core 1 (APP CPU) ([BT_BLUEDROID_PINNED_TO_CORE_1](#))

CONFIG_BT_BTU_TASK_STACK_SIZE

Bluetooth Bluedroid Host Stack task stack size

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#)

This select btu task stack size

CONFIG_BT_BLUEDROID_MEM_DEBUG

Bluedroid memory debug

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#)

Bluedroid memory debug

CONFIG_BT_CLASSIC_ENABLED

Classic Bluetooth

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#)

For now this option needs “[SMP_ENABLE](#)” to be set to yes

CONFIG_BT_A2DP_ENABLE

A2DP

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#) > [CONFIG_BT_CLASSIC_ENABLED](#)

Advanced Audio Distribution Profile

CONFIG_BT_SPP_ENABLED

SPP

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#) > [CONFIG_BT_CLASSIC_ENABLED](#)

This enables the Serial Port Profile

CONFIG_BT_HFP_ENABLE

Hands Free/Handset Profile

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#) > [CONFIG_BT_CLASSIC_ENABLED](#)

CONFIG_BT_HFP_ROLE

Hands-free Profile Role configuration

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#) > [CONFIG_BT_CLASSIC_ENABLED](#) > [CONFIG_BT_HFP_ENABLE](#)

Available options:

- Hands Free Unit (BT_HFP_CLIENT_ENABLE)
- Audio Gateway (BT_HFP_AG_ENABLE)

CONFIG_BT_HFP_AUDIO_DATA_PATH

audio(SCO) data path

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#) > [CONFIG_BT_CLASSIC_ENABLED](#) > [CONFIG_BT_HFP_ENABLE](#)

SCO data path, i.e. HCI or PCM. This option is set using API “esp_bredr_sco_datapath_set” in Bluetooth host. Default SCO data path can also be set in Bluetooth Controller.

Available options:

- PCM (BT_HFP_AUDIO_DATA_PATH_PCM)
- HCI (BT_HFP_AUDIO_DATA_PATH_HCI)

CONFIG_BT_HFP_WBS_ENABLE

Wide Band Speech

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#)

This enables Wide Band Speech. Should disable it when SCO data path is PCM. Otherwise there will be no data transmitted via GPIOs.

CONFIG_BT_SSP_ENABLED

Secure Simple Pairing

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#)

This enables the Secure Simple Pairing. If disable this option, Bluedroid will only support Legacy Pairing

CONFIG_BT_BLE_ENABLED

Bluetooth Low Energy

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#)

This enables Bluetooth Low Energy

CONFIG_BT_GATTS_ENABLE

Include GATT server module(GATTS)

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#) > [CONFIG_BT_BLE_ENABLED](#)

This option can be disabled when the app work only on gatt client mode

CONFIG_BT_GATTS_PPCP_CHAR_GAP

Enable Peripheral Preferred Connection Parameters characteristic in GAP service

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#) > [CONFIG_BT_BLE_ENABLED](#) > [CONFIG_BT_GATTS_ENABLE](#)

This enables “Peripheral Preferred Connection Parameters” characteristic (UUID: 0x2A04) in GAP service that has connection parameters like min/max connection interval, slave latency and supervision timeout multiplier

CONFIG_BT_BLE_BLUFI_ENABLE

Include blufi function

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#) > [CONFIG_BT_BLE_ENABLED](#) > [CONFIG_BT_GATTS_ENABLE](#)

This option can be close when the app does not require blufi function.

CONFIG_BT_GATTS_SEND_SERVICE_CHANGE_MODE

GATTS Service Change Mode

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#) > [CONFIG_BT_BLE_ENABLED](#) > [CONFIG_BT_GATTS_ENABLE](#)

Service change indication mode for GATT Server.

Available options:

- GATTS manually send service change indication (BT_GATTS_SEND_SERVICE_CHANGE_MANUAL)
Manually send service change indication through API `esp_ble_gatts_send_service_change_indication()`
- GATTS automatically send service change indication (BT_GATTS_SEND_SERVICE_CHANGE_AUTO)
Let Bluedroid handle the service change indication internally

CONFIG_BT_GATTC_ENABLE

Include GATT client module(GATTC)

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#) > [CONFIG_BT_BLE_ENABLED](#)

This option can be close when the app work only on gatt server mode

CONFIG_BT_GATTC_CACHE_NVS_FLASH

Save gattc cache data to nvs flash

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#) > [CONFIG_BT_BLE_ENABLED](#) > [CONFIG_BT_GATTC_ENABLE](#)

This select can save gattc cache data to nvs flash

CONFIG_BT_BLE_SMP_ENABLE

Include BLE security module(SMP)

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#) > [CONFIG_BT_BLE_ENABLED](#)

This option can be close when the app not used the ble security connect.

CONFIG_BT_SMP_SLAVE_CON_PARAMS_UPD_ENABLE

Slave enable connection parameters update during pairing

Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_BLE_ENABLED > CONFIG_BT_BLE_SMP_ENABLE

In order to reduce the pairing time, slave actively initiates connection parameters update during pairing.

CONFIG_BT_STACK_NO_LOG

Disable BT debug logs (minimize bin size)

Found in: Component config > Bluetooth > Bluedroid Options

This select can save the rodata code size

BT DEBUG LOG LEVEL Contains:

- *CONFIG_BT_LOG_HCI_TRACE_LEVEL*
- *CONFIG_BT_LOG_BTM_TRACE_LEVEL*
- *CONFIG_BT_LOG_L2CAP_TRACE_LEVEL*
- *CONFIG_BT_LOG_RFCOMM_TRACE_LEVEL*
- *CONFIG_BT_LOG_SDP_TRACE_LEVEL*
- *CONFIG_BT_LOG_GAP_TRACE_LEVEL*
- *CONFIG_BT_LOG_BNEP_TRACE_LEVEL*
- *CONFIG_BT_LOG_PAN_TRACE_LEVEL*
- *CONFIG_BT_LOG_A2D_TRACE_LEVEL*
- *CONFIG_BT_LOG_AVDT_TRACE_LEVEL*
- *CONFIG_BT_LOG_AVCT_TRACE_LEVEL*
- *CONFIG_BT_LOG_AVRC_TRACE_LEVEL*
- *CONFIG_BT_LOG_MCA_TRACE_LEVEL*
- *CONFIG_BT_LOG_HID_TRACE_LEVEL*
- *CONFIG_BT_LOG_APPL_TRACE_LEVEL*
- *CONFIG_BT_LOG_GATT_TRACE_LEVEL*
- *CONFIG_BT_LOG_SMP_TRACE_LEVEL*
- *CONFIG_BT_LOG_BTIF_TRACE_LEVEL*
- *CONFIG_BT_LOG_BTC_TRACE_LEVEL*
- *CONFIG_BT_LOG_OSI_TRACE_LEVEL*
- *CONFIG_BT_LOG_BLUFI_TRACE_LEVEL*

CONFIG_BT_LOG_HCI_TRACE_LEVEL

HCI layer

Found in: Component config > Bluetooth > Bluedroid Options > BT DEBUG LOG LEVEL

Define BT trace level for HCI layer

Available options:

- NONE (BT_LOG_HCI_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_HCI_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_HCI_TRACE_LEVEL_WARNING)
- API (BT_LOG_HCI_TRACE_LEVEL_API)
- EVENT (BT_LOG_HCI_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_HCI_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_HCI_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_BTM_TRACE_LEVEL

BTM layer

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#) > [BT DEBUG LOG LEVEL](#)

Define BT trace level for BTM layer

Available options:

- NONE (BT_LOG_BTM_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_BTM_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_BTM_TRACE_LEVEL_WARNING)
- API (BT_LOG_BTM_TRACE_LEVEL_API)
- EVENT (BT_LOG_BTM_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_BTM_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_BTM_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_L2CAP_TRACE_LEVEL

L2CAP layer

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#) > [BT DEBUG LOG LEVEL](#)

Define BT trace level for L2CAP layer

Available options:

- NONE (BT_LOG_L2CAP_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_L2CAP_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_L2CAP_TRACE_LEVEL_WARNING)
- API (BT_LOG_L2CAP_TRACE_LEVEL_API)
- EVENT (BT_LOG_L2CAP_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_L2CAP_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_L2CAP_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_RFCOMM_TRACE_LEVEL

RFCOMM layer

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#) > [BT DEBUG LOG LEVEL](#)

Define BT trace level for RFCOMM layer

Available options:

- NONE (BT_LOG_RFCOMM_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_RFCOMM_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_RFCOMM_TRACE_LEVEL_WARNING)
- API (BT_LOG_RFCOMM_TRACE_LEVEL_API)
- EVENT (BT_LOG_RFCOMM_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_RFCOMM_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_RFCOMM_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_SDP_TRACE_LEVEL

SDP layer

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#) > [BT DEBUG LOG LEVEL](#)

Define BT trace level for SDP layer

Available options:

- NONE (BT_LOG_SDP_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_SDP_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_SDP_TRACE_LEVEL_WARNING)
- API (BT_LOG_SDP_TRACE_LEVEL_API)

- EVENT (BT_LOG_SDP_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_SDP_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_SDP_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_GAP_TRACE_LEVEL

GAP layer

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#) > [BT DEBUG LOG LEVEL](#)

Define BT trace level for GAP layer

Available options:

- NONE (BT_LOG_GAP_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_GAP_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_GAP_TRACE_LEVEL_WARNING)
- API (BT_LOG_GAP_TRACE_LEVEL_API)
- EVENT (BT_LOG_GAP_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_GAP_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_GAP_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_BNEP_TRACE_LEVEL

BNEP layer

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#) > [BT DEBUG LOG LEVEL](#)

Define BT trace level for BNEP layer

Available options:

- NONE (BT_LOG_BNEP_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_BNEP_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_BNEP_TRACE_LEVEL_WARNING)
- API (BT_LOG_BNEP_TRACE_LEVEL_API)
- EVENT (BT_LOG_BNEP_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_BNEP_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_BNEP_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_PAN_TRACE_LEVEL

PAN layer

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#) > [BT DEBUG LOG LEVEL](#)

Define BT trace level for PAN layer

Available options:

- NONE (BT_LOG_PAN_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_PAN_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_PAN_TRACE_LEVEL_WARNING)
- API (BT_LOG_PAN_TRACE_LEVEL_API)
- EVENT (BT_LOG_PAN_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_PAN_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_PAN_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_A2D_TRACE_LEVEL

A2D layer

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#) > [BT DEBUG LOG LEVEL](#)

Define BT trace level for A2D layer

Available options:

- NONE (BT_LOG_A2D_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_A2D_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_A2D_TRACE_LEVEL_WARNING)
- API (BT_LOG_A2D_TRACE_LEVEL_API)
- EVENT (BT_LOG_A2D_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_A2D_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_A2D_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_AVDT_TRACE_LEVEL

AVDT layer

Found in: [Component config](#) > [Bluetooth](#) > [Blueroid Options](#) > [BT DEBUG LOG LEVEL](#)

Define BT trace level for AVDT layer

Available options:

- NONE (BT_LOG_AVDT_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_AVDT_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_AVDT_TRACE_LEVEL_WARNING)
- API (BT_LOG_AVDT_TRACE_LEVEL_API)
- EVENT (BT_LOG_AVDT_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_AVDT_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_AVDT_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_AVCT_TRACE_LEVEL

AVCT layer

Found in: [Component config](#) > [Bluetooth](#) > [Blueroid Options](#) > [BT DEBUG LOG LEVEL](#)

Define BT trace level for AVCT layer

Available options:

- NONE (BT_LOG_AVCT_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_AVCT_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_AVCT_TRACE_LEVEL_WARNING)
- API (BT_LOG_AVCT_TRACE_LEVEL_API)
- EVENT (BT_LOG_AVCT_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_AVCT_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_AVCT_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_AVRC_TRACE_LEVEL

AVRC layer

Found in: [Component config](#) > [Bluetooth](#) > [Blueroid Options](#) > [BT DEBUG LOG LEVEL](#)

Define BT trace level for AVRC layer

Available options:

- NONE (BT_LOG_AVRC_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_AVRC_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_AVRC_TRACE_LEVEL_WARNING)
- API (BT_LOG_AVRC_TRACE_LEVEL_API)
- EVENT (BT_LOG_AVRC_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_AVRC_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_AVRC_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_MCA_TRACE_LEVEL

MCA layer

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#) > [BT DEBUG LOG LEVEL](#)

Define BT trace level for MCA layer

Available options:

- NONE (BT_LOG_MCA_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_MCA_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_MCA_TRACE_LEVEL_WARNING)
- API (BT_LOG_MCA_TRACE_LEVEL_API)
- EVENT (BT_LOG_MCA_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_MCA_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_MCA_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_HID_TRACE_LEVEL

HID layer

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#) > [BT DEBUG LOG LEVEL](#)

Define BT trace level for HID layer

Available options:

- NONE (BT_LOG_HID_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_HID_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_HID_TRACE_LEVEL_WARNING)
- API (BT_LOG_HID_TRACE_LEVEL_API)
- EVENT (BT_LOG_HID_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_HID_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_HID_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_APPL_TRACE_LEVEL

APPL layer

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#) > [BT DEBUG LOG LEVEL](#)

Define BT trace level for APPL layer

Available options:

- NONE (BT_LOG_APPL_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_APPL_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_APPL_TRACE_LEVEL_WARNING)
- API (BT_LOG_APPL_TRACE_LEVEL_API)
- EVENT (BT_LOG_APPL_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_APPL_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_APPL_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_GATT_TRACE_LEVEL

GATT layer

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#) > [BT DEBUG LOG LEVEL](#)

Define BT trace level for GATT layer

Available options:

- NONE (BT_LOG_GATT_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_GATT_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_GATT_TRACE_LEVEL_WARNING)
- API (BT_LOG_GATT_TRACE_LEVEL_API)

- EVENT (BT_LOG_GATT_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_GATT_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_GATT_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_SMP_TRACE_LEVEL

SMP layer

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#) > [BT DEBUG LOG LEVEL](#)

Define BT trace level for SMP layer

Available options:

- NONE (BT_LOG_SMP_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_SMP_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_SMP_TRACE_LEVEL_WARNING)
- API (BT_LOG_SMP_TRACE_LEVEL_API)
- EVENT (BT_LOG_SMP_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_SMP_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_SMP_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_BTIF_TRACE_LEVEL

BTIF layer

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#) > [BT DEBUG LOG LEVEL](#)

Define BT trace level for BTIF layer

Available options:

- NONE (BT_LOG_BTIF_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_BTIF_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_BTIF_TRACE_LEVEL_WARNING)
- API (BT_LOG_BTIF_TRACE_LEVEL_API)
- EVENT (BT_LOG_BTIF_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_BTIF_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_BTIF_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_BTC_TRACE_LEVEL

BTC layer

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#) > [BT DEBUG LOG LEVEL](#)

Define BT trace level for BTC layer

Available options:

- NONE (BT_LOG_BTC_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_BTC_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_BTC_TRACE_LEVEL_WARNING)
- API (BT_LOG_BTC_TRACE_LEVEL_API)
- EVENT (BT_LOG_BTC_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_BTC_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_BTC_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_OSI_TRACE_LEVEL

OSI layer

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#) > [BT DEBUG LOG LEVEL](#)

Define BT trace level for OSI layer

Available options:

- NONE (BT_LOG_OSI_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_OSI_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_OSI_TRACE_LEVEL_WARNING)
- API (BT_LOG_OSI_TRACE_LEVEL_API)
- EVENT (BT_LOG_OSI_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_OSI_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_OSI_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_BLUFI_TRACE_LEVEL

BLUFI layer

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#) > [BT DEBUG LOG LEVEL](#)

Define BT trace level for BLUFI layer

Available options:

- NONE (BT_LOG_BLUFI_TRACE_LEVEL_NONE)
- ERROR (BT_LOG_BLUFI_TRACE_LEVEL_ERROR)
- WARNING (BT_LOG_BLUFI_TRACE_LEVEL_WARNING)
- API (BT_LOG_BLUFI_TRACE_LEVEL_API)
- EVENT (BT_LOG_BLUFI_TRACE_LEVEL_EVENT)
- DEBUG (BT_LOG_BLUFI_TRACE_LEVEL_DEBUG)
- VERBOSE (BT_LOG_BLUFI_TRACE_LEVEL_VERBOSE)

CONFIG_BT_ACL_CONNECTIONS

BT/BLE MAX ACL CONNECTIONS(1~7)

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#)

Maximum BT/BLE connection count

CONFIG_BT_ALLOCATION_FROM_SPIRAM_FIRST

BT/BLE will first malloc the memory from the PSRAM

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#)

This select can save the internal RAM if there have the PSRAM

CONFIG_BT_BLE_DYNAMIC_ENV_MEMORY

Use dynamic memory allocation in BT/BLE stack

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#)

This select can make the allocation of memory will become more flexible

CONFIG_BT_BLE_HOST_QUEUE_CONG_CHECK

BLE queue congestion check

Found in: [Component config](#) > [Bluetooth](#) > [Bluedroid Options](#)

When scanning and scan duplicate is not enabled, if there are a lot of adv packets around or application layer handling adv packets is slow, it will cause the controller memory to run out. if enabled, adv packets will be lost when host queue is congested.

CONFIG_BT_BLE_ACT_SCAN_REP_ADV_SCAN

Report adv data and scan response individually when BLE active scan

Found in: Component config > Bluetooth > Bluedroid Options

Originally, when doing BLE active scan, Bluedroid will not report adv to application layer until receive scan response. This option is used to disable the behavior. When enable this option, Bluedroid will report adv data or scan response to application layer immediately.

Memory reserved at start of DRAM for Bluetooth stack

CONFIG_BT_BLE_ESTAB_LINK_CONN_TOUT

Timeout of BLE connection establishment

Found in: Component config > Bluetooth > Bluedroid Options

Bluetooth Connection establishment maximum time, if connection time exceeds this value, the connection establishment fails, ESP_GATTC_OPEN_EVT or ESP_GATTS_OPEN_EVT is triggered.

NimBLE Options Contains:

- CONFIG_BT_NIMBLE_MEM_ALLOC_MODE
- CONFIG_BT_NIMBLE_MAX_CONNECTIONS
- CONFIG_BT_NIMBLE_MAX_BONDS
- CONFIG_BT_NIMBLE_MAX_CCCDS
- CONFIG_BT_NIMBLE_L2CAP_COC_MAX_NUM
- CONFIG_BT_NIMBLE_PINNED_TO_CORE_CHOICE
- CONFIG_BT_NIMBLE_TASK_STACK_SIZE
- CONFIG_BT_NIMBLE_ROLE_CENTRAL
- CONFIG_BT_NIMBLE_ROLE_PERIPHERAL
- CONFIG_BT_NIMBLE_ROLE_BROADCASTER
- CONFIG_BT_NIMBLE_ROLE_OBSERVER
- CONFIG_BT_NIMBLE_NVS_PERSIST
- CONFIG_BT_NIMBLE_SM_LEGACY
- CONFIG_BT_NIMBLE_SM_SC
- CONFIG_BT_NIMBLE_DEBUG
- CONFIG_BT_NIMBLE_SM_SC_DEBUG_KEYS
- CONFIG_BT_NIMBLE_SVC_GAP_DEVICE_NAME
- CONFIG_BT_NIMBLE_GAP_DEVICE_NAME_MAX_LEN
- CONFIG_BT_NIMBLE_ATT_PREFERRED_MTU
- CONFIG_BT_NIMBLE_SVC_GAP_APPEARANCE
- CONFIG_BT_NIMBLE_ACL_BUF_COUNT
- CONFIG_BT_NIMBLE_ACL_BUF_SIZE
- CONFIG_BT_NIMBLE_HCI_EVT_BUF_SIZE
- CONFIG_BT_NIMBLE_HCI_EVT_HI_BUF_COUNT
- CONFIG_BT_NIMBLE_HCI_EVT_LO_BUF_COUNT
- CONFIG_BT_NIMBLE_MSYS1_BLOCK_COUNT
- CONFIG_BT_NIMBLE_HS_FLOW_CTRL
- CONFIG_BT_NIMBLE_RPA_TIMEOUT
- CONFIG_BT_NIMBLE_MESH
- CONFIG_BT_NIMBLE_CRYPTO_STACK_MBEDTLS

CONFIG_BT_NIMBLE_MEM_ALLOC_MODE

Memory allocation strategy

Found in: Component config > Bluetooth > NimBLE Options

Allocation strategy for NimBLE host stack, essentially provides ability to allocate all required dynamic allocations from,

- Internal DRAM memory only
- External SPIRAM memory only
- Either internal or external memory based on default malloc() behavior in ESP-IDF

Recommended mode here is always internal, since that is most preferred from security perspective. But if application requirement does not allow sufficient free internal memory then alternate mode can be selected.

Available options:

- Internal memory (BT_NIMBLE_MEM_ALLOC_MODE_INTERNAL)
- External SPIRAM (BT_NIMBLE_MEM_ALLOC_MODE_EXTERNAL)
- Default alloc mode (BT_NIMBLE_MEM_ALLOC_MODE_DEFAULT)

CONFIG_BT_NIMBLE_MAX_CONNECTIONS

Maximum number of concurrent connections

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#)

Defines maximum number of concurrent BLE connections

CONFIG_BT_NIMBLE_MAX BONDS

Maximum number of bonds to save across reboots

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#)

Defines maximum number of bonds to save for peer security and our security

CONFIG_BT_NIMBLE_MAX_CCCDS

Maximum number of CCC descriptors to save across reboots

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#)

Defines maximum number of CCC descriptors to save

CONFIG_BT_NIMBLE_L2CAP_COC_MAX_NUM

Maximum number of connection oriented channels

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#)

Defines maximum number of BLE Connection Oriented Channels. When set to (0), BLE COC is not compiled in

CONFIG_BT_NIMBLE_PINNED_TO_CORE_CHOICE

The CPU core on which NimBLE host will run

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#)

The CPU core on which NimBLE host will run. You can choose Core 0 or Core 1. Cannot specify no-affinity

Available options:

- Core 0 (PRO CPU) (BT_NIMBLE_PINNED_TO_CORE_0)
- Core 1 (APP CPU) (BT_NIMBLE_PINNED_TO_CORE_1)

CONFIG_BT_NIMBLE_TASK_STACK_SIZE

NimBLE Host task stack size

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#)

This configures stack size of NimBLE host task

CONFIG_BT_NIMBLE_ROLE_CENTRAL

Enable BLE Central role

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#)

CONFIG_BT_NIMBLE_ROLE_PERIPHERAL

Enable BLE Peripheral role

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#)

CONFIG_BT_NIMBLE_ROLE_BROADCASTER

Enable BLE Broadcaster role

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#)

CONFIG_BT_NIMBLE_ROLE_OBSERVER

Enable BLE Observer role

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#)

CONFIG_BT_NIMBLE_NVS_PERSIST

Persist the BLE Bonding keys in NVS

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#)

Enable this flag to make bonding persistent across device reboots

CONFIG_BT_NIMBLE_SM_LEGACY

Security manager legacy pairing

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#)

Enable security manager legacy pairing

CONFIG_BT_NIMBLE_SM_SC

Security manager secure connections (4.2)

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#)

Enable security manager secure connections

CONFIG_BT_NIMBLE_DEBUG

Enable extra runtime asserts and host debugging

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#)

This enables extra runtime asserts and host debugging

CONFIG_BT_NIMBLE_SM_SC_DEBUG_KEYS

Use predefined public-private key pair

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#)

If this option is enabled, SM uses predefined DH key pair as described in Core Specification, Vol. 3, Part H, 2.3.5.6.1. This allows to decrypt air traffic easily and thus should only be used for debugging.

CONFIG_BT_NIMBLE_SVC_GAP_DEVICE_NAME

BLE GAP default device name

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#)

The Device Name characteristic shall contain the name of the device as an UTF-8 string. This name can be changed by using API `ble_svc_gap_device_name_set()`

CONFIG_BT_NIMBLE_GAP_DEVICE_NAME_MAX_LEN

Maximum length of BLE device name in octets

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#)

Device Name characteristic value shall be 0 to 248 octets in length

CONFIG_BT_NIMBLE_ATT_PREFERRED_MTU

Preferred MTU size in octets

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#)

This is the default value of ATT MTU indicated by the device during an ATT MTU exchange. This value can be changed using API `ble_att_set_preferred_mtu()`

CONFIG_BT_NIMBLE_SVC_GAP_APPEARANCE

External appearance of the device

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#)

Standard BLE GAP Appearance value in HEX format e.g. 0x02C0

CONFIG_BT_NIMBLE_ACL_BUF_COUNT

ACL Buffer count

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#)

The number of ACL data buffers.

CONFIG_BT_NIMBLE_ACL_BUF_SIZE

ACL Buffer size

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#)

This is the maximum size of the data portion of HCI ACL data packets. It does not include the HCI data header (of 4 bytes)

CONFIG_BT_NIMBLE_HCI_EVT_BUF_SIZE

HCI Event Buffer size

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#)

This is the size of each HCI event buffer in bytes

CONFIG_BT_NIMBLE_HCI_EVT_HI_BUF_COUNT

High Priority HCI Event Buffer count

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#)

This is the high priority HCI events' buffer size. High-priority event buffers are for everything except advertising reports. If there are no free high-priority event buffers then host will try to allocate a low-priority buffer instead

CONFIG_BT_NIMBLE_HCI_EVT_LO_BUF_COUNT

Low Priority HCI Event Buffer count

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#)

This is the low priority HCI events' buffer size. Low-priority event buffers are only used for advertising reports. If there are no free low-priority event buffers, then an incoming advertising report will get dropped

CONFIG_BT_NIMBLE_MSYS1_BLOCK_COUNT

MSYS_1 Block Count

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#)

MSYS is a system level mbuf registry. For prepare write & prepare responses Mbufs are allocated out of msys_1 pool. For NIMBLE_MESH enabled cases, this block count is increased by 8 than user defined count.

CONFIG_BT_NIMBLE_HS_FLOW_CTRL

Enable Host Flow control

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#)

Enable Host Flow control

CONFIG_BT_NIMBLE_HS_FLOW_CTRL_ITVL

Host Flow control interval

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#) > [CONFIG_BT_NIMBLE_HS_FLOW_CTRL](#)

Host flow control interval in msecs

CONFIG_BT_NIMBLE_HS_FLOW_CTRL_THRESH

Host Flow control threshold

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#) > [CONFIG_BT_NIMBLE_HS_FLOW_CTRL](#)

Host flow control threshold, if the number of free buffers are at or below this threshold, send an immediate number-of-completed-packets event

CONFIG_BT_NIMBLE_HS_FLOW_CTRL_TX_ON_DISCONNECT

Host Flow control on disconnect

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#) > [CONFIG_BT_NIMBLE_HS_FLOW_CTRL](#)

Enable this option to send number-of-completed-packets event to controller after disconnection

CONFIG_BT_NIMBLE_RPA_TIMEOUT

RPA timeout in seconds

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#)

Time interval between RPA address change. This is applicable in case of Host based RPA

CONFIG_BT_NIMBLE_MESH

Enable BLE mesh functionality

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#)

Enable BLE Mesh functionality

Contains:

- [CONFIG_BT_NIMBLE_MESH_PROXY](#)
- [CONFIG_BT_NIMBLE_MESH_PROV](#)
- [CONFIG_BT_NIMBLE_MESH_GATT_PROXY](#)
- [CONFIG_BT_NIMBLE_MESH_RELAY](#)
- [CONFIG_BT_NIMBLE_MESH_LOW_POWER](#)
- [CONFIG_BT_NIMBLE_MESH_FRIEND](#)
- [CONFIG_BT_NIMBLE_MESH_DEVICE_NAME](#)

CONFIG_BT_NIMBLE_MESH_PROXY

Enable mesh proxy functionality

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#) > [CONFIG_BT_NIMBLE_MESH](#)

Enable proxy. This is automatically set whenever NIMBLE_MESH_PB_GATT or NIMBLE_MESH_GATT_PROXY is set

CONFIG_BT_NIMBLE_MESH_PROV

Enable BLE mesh provisioning

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#) > [CONFIG_BT_NIMBLE_MESH](#)

Enable mesh provisioning

CONFIG_BT_NIMBLE_MESH_PB_ADV

Enable mesh provisioning over advertising bearer

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#) > [CONFIG_BT_NIMBLE_MESH](#) > [CONFIG_BT_NIMBLE_MESH_PROV](#)

Enable this option to allow the device to be provisioned over the advertising bearer

CONFIG_BT_NIMBLE_MESH_PB_GATT

Enable mesh provisioning over GATT bearer

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#) > [CONFIG_BT_NIMBLE_MESH](#) > [CONFIG_BT_NIMBLE_MESH_PROV](#)

Enable this option to allow the device to be provisioned over the GATT bearer

CONFIG_BT_NIMBLE_MESH_GATT_PROXY

Enable GATT Proxy functionality

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#) > [CONFIG_BT_NIMBLE_MESH](#)

This option enables support for the Mesh GATT Proxy Service, i.e. the ability to act as a proxy between a Mesh GATT Client and a Mesh network

CONFIG_BT_NIMBLE_MESH_RELAY

Enable mesh relay functionality

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#) > [CONFIG_BT_NIMBLE_MESH](#)

Support for acting as a Mesh Relay Node

CONFIG_BT_NIMBLE_MESH_LOW_POWER

Enable mesh low power mode

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#) > [CONFIG_BT_NIMBLE_MESH](#)

Enable this option to be able to act as a Low Power Node

CONFIG_BT_NIMBLE_MESH_FRIEND

Enable mesh friend functionality

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#) > [CONFIG_BT_NIMBLE_MESH](#)

Enable this option to be able to act as a Friend Node

CONFIG_BT_NIMBLE_MESH_DEVICE_NAME

Set mesh device name

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#) > [CONFIG_BT_NIMBLE_MESH](#)

This value defines Bluetooth Mesh device/node name

CONFIG_BT_NIMBLE_CRYPTOSTACK_MBEDTLS

Override TinyCrypt with mbedTLS for crypto computations

Found in: [Component config](#) > [Bluetooth](#) > [NimBLE Options](#)

Enable this option to choose mbedTLS instead of TinyCrypt for crypto computations.

CONFIG_BLE_MESH

ESP BLE Mesh Support

Found in: Component config

This option enables ESP BLE Mesh support. The specific features that are available may depend on other features that have been enabled in the stack, such as Bluetooth Support, Bluedroid Support & GATT support.

Contains:

- *CONFIG_BLE_MESH_HCI_5_0*
- *CONFIG_BLE_MESH_USE_DUPLICATE_SCAN*
- *CONFIG_BLE_MESH_MEM_ALLOC_MODE*
- *CONFIG_BLE_MESH_FREERTOS_STATIC_ALLOC*
- *CONFIG_BLE_MESH_DEINIT*
- *BLE Mesh and BLE coexistence support*
- *CONFIG_BLE_MESH_FAST_PROV*
- *CONFIG_BLE_MESH_NODE*
- *CONFIG_BLE_MESH_PROVISIONER*
- *CONFIG_BLE_MESH_PROV*
- *CONFIG_BLE_MESH_PB_ADV*
- *CONFIG_BLE_MESH_PB_GATT*
- *CONFIG_BLE_MESH_PROXY*
- *CONFIG_BLE_MESH_GATT_PROXY_SERVER*
- *CONFIG_BLE_MESH_GATT_PROXY_CLIENT*
- *CONFIG_BLE_MESH_SETTINGS*
- *CONFIG_BLE_MESH_SUBNET_COUNT*
- *CONFIG_BLE_MESH_APP_KEY_COUNT*
- *CONFIG_BLE_MESH_MODEL_KEY_COUNT*
- *CONFIG_BLE_MESH_MODEL_GROUP_COUNT*
- *CONFIG_BLE_MESH_LABEL_COUNT*
- *CONFIG_BLE_MESH_CRPL*
- *CONFIG_BLE_MESH_MSG_CACHE_SIZE*
- *CONFIG_BLE_MESH_ADV_BUF_COUNT*
- *CONFIG_BLE_MESH_IVU_DIVIDER*
- *CONFIG_BLE_MESH_TX_SEG_MSG_COUNT*
- *CONFIG_BLE_MESH_RX_SEG_MSG_COUNT*
- *CONFIG_BLE_MESH_RX_SDU_MAX*
- *CONFIG_BLE_MESH_TX_SEG_MAX*
- *CONFIG_BLE_MESH_RELAY*
- *CONFIG_BLE_MESH_LOW_POWER*
- *CONFIG_BLE_MESH_FRIEND*
- *CONFIG_BLE_MESH_NO_LOG*
- *BLE Mesh STACK DEBUG LOG LEVEL*
- *BLE Mesh NET BUF DEBUG LOG LEVEL*
- *CONFIG_BLE_MESH_CLIENT_MSG_TIMEOUT*
- *Support for BLE Mesh Foundation models*
- *Support for BLE Mesh Client/Server models*
- *CONFIG_BLE_MESH_IV_UPDATE_TEST*
- *BLE Mesh specific test option*

CONFIG_BLE_MESH_HCI_5_0

Support sending 20ms non-connectable adv packets

Found in: Component config > CONFIG_BLE_MESH

It is a temporary solution and needs further modifications.

CONFIG_BLE_MESH_USE_DUPLICATE_SCAN

Support Duplicate Scan in BLE Mesh

Found in: *Component config* > *CONFIG_BLE_MESH*

Enable this option to allow using specific duplicate scan filter in BLE Mesh, and Scan Duplicate Type must be set by choosing the option in the Bluetooth Controller section in menuconfig, which is “Scan Duplicate By Device Address and Advertising Data” .

CONFIG_BLE_MESH_MEM_ALLOC_MODE

Memory allocation strategy

Found in: *Component config* > *CONFIG_BLE_MESH*

Allocation strategy for BLE Mesh stack, essentially provides ability to allocate all required dynamic allocations from,

- Internal DRAM memory only
- External SPIRAM memory only
- Either internal or external memory based on default malloc() behavior in ESP-IDF

Recommended mode here is always internal, since that is most preferred from security perspective. But if application requirement does not allow sufficient free internal memory then alternate mode can be selected.

Available options:

- Internal DRAM (BLE_MESH_MEM_ALLOC_MODE_INTERNAL)
- External SPIRAM (BLE_MESH_MEM_ALLOC_MODE_EXTERNAL)
- Default alloc mode (BLE_MESH_MEM_ALLOC_MODE_DEFAULT)
Enable this option to use the default memory allocation strategy when external SPIRAM is enabled. See the SPIRAM options for more details.

CONFIG_BLE_MESH_FREERTOS_STATIC_ALLOC

Enable FreeRTOS static allocation

Found in: *Component config* > *CONFIG_BLE_MESH*

Enable this option to use FreeRTOS static allocation APIs for BLE Mesh, which provides the ability to use different dynamic memory (i.e. SPIRAM) for FreeRTOS objects. If this option is disabled, the FreeRTOS static allocation APIs will not be used, and internal DRAM will be allocated for FreeRTOS objects.

CONFIG_BLE_MESH_FREERTOS_STATIC_ALLOC_MODE

Memory allocation for FreeRTOS objects

Found in: *Component config* > *CONFIG_BLE_MESH* > *CONFIG_BLE_MESH_FREERTOS_STATIC_ALLOC*

Choose the memory to be used for FreeRTOS objects.

Available options:

- External SPIRAM (BLE_MESH_FREERTOS_STATIC_ALLOC_EXTERNAL)
If enabled, BLE Mesh allocates dynamic memory from external SPIRAM for FreeRTOS objects, i.e. mutex, queue, and task stack. External SPIRAM can only be used for task stack when SPIRAM_ALLOW_STACK_EXTERNAL_MEMORY is enabled. See the SPIRAM options for more details.

CONFIG_BLE_MESH_DEINIT

Support de-initialize BLE Mesh stack

Found in: [Component config](#) > [CONFIG_BLE_MESH](#)

If enabled, users can use the function `esp_ble_mesh_deinit()` to de-initialize the whole BLE Mesh stack.

BLE Mesh and BLE coexistence support Contains:

- [CONFIG_BLE_MESH_SUPPORT_BLE_ADV](#)
- [CONFIG_BLE_MESH_SUPPORT_BLE_SCAN](#)

CONFIG_BLE_MESH_SUPPORT_BLE_ADV

Support sending normal BLE advertising packets

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [BLE Mesh and BLE coexistence support](#)

When selected, users can send normal BLE advertising packets with specific API.

CONFIG_BLE_MESH_BLE_ADV_BUF_COUNT

Number of advertising buffers for BLE advertising packets

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [BLE Mesh and BLE coexistence support](#) > [CONFIG_BLE_MESH_SUPPORT_BLE_ADV](#)

Number of advertising buffers for BLE packets available.

CONFIG_BLE_MESH_SUPPORT_BLE_SCAN

Support scanning normal BLE advertising packets

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [BLE Mesh and BLE coexistence support](#)

When selected, users can register a callback and receive normal BLE advertising packets in the application layer.

CONFIG_BLE_MESH_FAST_PROV

Enable BLE Mesh Fast Provisioning

Found in: [Component config](#) > [CONFIG_BLE_MESH](#)

Enable this option to allow BLE Mesh fast provisioning solution to be used. When there are multiple unprovisioned devices around, fast provisioning can greatly reduce the time consumption of the whole provisioning process. When this option is enabled, and after an unprovisioned device is provisioned into a node successfully, it can be changed to a temporary Provisioner.

CONFIG_BLE_MESH_NODE

Support for BLE Mesh Node

Found in: [Component config](#) > [CONFIG_BLE_MESH](#)

Enable the device to be provisioned into a node. This option should be enabled when an unprovisioned device is going to be provisioned into a node and communicate with other nodes in the BLE Mesh network.

CONFIG_BLE_MESH_PROVISIONER

Support for BLE Mesh Provisioner

Found in: [Component config](#) > [CONFIG_BLE_MESH](#)

Enable the device to be a Provisioner. The option should be enabled when a device is going to act as a Provisioner and provision unprovisioned devices into the BLE Mesh network.

CONFIG_BLE_MESH_WAIT_FOR_PROV_MAX_DEV_NUM

Maximum number of unprovisioned devices that can be added to device queue

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_PROVISIONER](#)

This option specifies how many unprovisioned devices can be added to device queue for provisioning. Users can use this option to define the size of the queue in the bottom layer which is used to store unprovisioned device information (e.g. Device UUID, address).

CONFIG_BLE_MESH_MAX_PROV_NODES

Maximum number of devices that can be provisioned by Provisioner

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_PROVISIONER](#)

This option specifies how many devices can be provisioned by a Provisioner. This value indicates the maximum number of unprovisioned devices which can be provisioned by a Provisioner. For instance, if the value is 6, it means the Provisioner can provision up to 6 unprovisioned devices. Theoretically a Provisioner without the limitation of its memory can provision up to 32766 unprovisioned devices, here we limit the maximum number to 100 just to limit the memory used by a Provisioner. The bigger the value is, the more memory it will cost by a Provisioner to store the information of nodes.

CONFIG_BLE_MESH_PBA_SAME_TIME

Maximum number of PB-ADV running at the same time by Provisioner

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_PROVISIONER](#)

This option specifies how many devices can be provisioned at the same time using PB-ADV. For examples, if the value is 2, it means a Provisioner can provision two unprovisioned devices with PB-ADV at the same time.

CONFIG_BLE_MESH_PBG_SAME_TIME

Maximum number of PB-GATT running at the same time by Provisioner

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_PROVISIONER](#)

This option specifies how many devices can be provisioned at the same time using PB-GATT. For example, if the value is 2, it means a Provisioner can provision two unprovisioned devices with PB-GATT at the same time.

CONFIG_BLE_MESH_PROVISIONER_SUBNET_COUNT

Maximum number of mesh subnets that can be created by Provisioner

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_PROVISIONER](#)

This option specifies how many subnets per network a Provisioner can create. Indeed, this value decides the number of network keys which can be added by a Provisioner.

CONFIG_BLE_MESH_PROVISIONER_APP_KEY_COUNT

Maximum number of application keys that can be owned by Provisioner

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_PROVISIONER](#)

This option specifies how many application keys the Provisioner can have. Indeed, this value decides the number of the application keys which can be added by a Provisioner.

CONFIG_BLE_MESH_PROVISIONER_RECV_HB

Support receiving Heartbeat messages

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_PROVISIONER](#)

When this option is enabled, Provisioner can call specific functions to enable or disable receiving Heartbeat messages and notify them to the application layer.

CONFIG_BLE_MESH_PROVISIONER_RECV_HB_FILTER_SIZE

Maximum number of filter entries for receiving Heartbeat messages

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_PROVISIONER](#) > [CONFIG_BLE_MESH_PROVISIONER_RECV_HB](#)

This option specifies how many heartbeat filter entries Provisioner supports. The heartbeat filter (acceptlist or rejectlist) entries are used to store a list of SRC and DST which can be used to decide if a heartbeat message will be processed and notified to the application layer by Provisioner. Note: The filter is an empty rejectlist by default.

CONFIG_BLE_MESH_PROV

BLE Mesh Provisioning support

Found in: [Component config](#) > [CONFIG_BLE_MESH](#)

Enable this option to support BLE Mesh Provisioning functionality. For BLE Mesh, this option should be always enabled.

CONFIG_BLE_MESH_PB_ADV

Provisioning support using the advertising bearer (PB-ADV)

Found in: [Component config](#) > [CONFIG_BLE_MESH](#)

Enable this option to allow the device to be provisioned over the advertising bearer. This option should be enabled if PB-ADV is going to be used during provisioning procedure.

CONFIG_BLE_MESH_UNPROVISIONED_BEACON_INTERVAL

Interval between two consecutive Unprovisioned Device Beacon

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_PB_ADV](#)

This option specifies the interval of sending two consecutive unprovisioned device beacon, users can use this option to change the frequency of sending unprovisioned device beacon. For example, if the value is 5, it means the unprovisioned device beacon will send every 5 seconds. When the option of BLE_MESH_FAST_PROV is selected, the value is better to be 3 seconds, or less.

CONFIG_BLE_MESH_PB_GATT

Provisioning support using GATT (PB-GATT)

Found in: [Component config](#) > [CONFIG_BLE_MESH](#)

Enable this option to allow the device to be provisioned over GATT. This option should be enabled if PB-GATT is going to be used during provisioning procedure.

Virtual option enabled whenever any Proxy protocol is needed

CONFIG_BLE_MESH_PROXY

BLE Mesh Proxy protocol support

Found in: [Component config](#) > [CONFIG_BLE_MESH](#)

Enable this option to support BLE Mesh Proxy protocol used by PB-GATT and other proxy pdu transmission.

CONFIG_BLE_MESH_GATT_PROXY_SERVER

BLE Mesh GATT Proxy Server

Found in: [Component config](#) > [CONFIG_BLE_MESH](#)

This option enables support for Mesh GATT Proxy Service, i.e. the ability to act as a proxy between a Mesh GATT Client and a Mesh network. This option should be enabled if a node is going to be a Proxy Server.

CONFIG_BLE_MESH_NODE_ID_TIMEOUT

Node Identity advertising timeout

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_GATT_PROXY_SERVER](#)

This option determines for how long the local node advertises using Node Identity. The given value is in seconds. The specification limits this to 60 seconds and lists it as the recommended value as well. So leaving the default value is the safest option. When an unprovisioned device is provisioned successfully and becomes a node, it will start to advertise using Node Identity during the time set by this option. And after that, Network ID will be advertised.

CONFIG_BLE_MESH_PROXY_FILTER_SIZE

Maximum number of filter entries per Proxy Client

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_GATT_PROXY_SERVER](#)

This option specifies how many Proxy Filter entries the local node supports. The entries of Proxy filter (whitelist or blacklist) are used to store a list of addresses which can be used to decide which messages will be forwarded to the Proxy Client by the Proxy Server.

CONFIG_BLE_MESH_GATT_PROXY_CLIENT

BLE Mesh GATT Proxy Client

Found in: [Component config](#) > [CONFIG_BLE_MESH](#)

This option enables support for Mesh GATT Proxy Client. The Proxy Client can use the GATT bearer to send mesh messages to a node that supports the advertising bearer.

CONFIG_BLE_MESH_SETTINGS

Store BLE Mesh configuration persistently

Found in: [Component config](#) > [CONFIG_BLE_MESH](#)

When selected, the BLE Mesh stack will take care of storing/restoring the BLE Mesh configuration persistently in flash. If the device is a BLE Mesh node, when this option is enabled, the configuration of the device will be stored persistently, including unicast address, NetKey, AppKey, etc. And if the device is a BLE Mesh Provisioner, the information of the device will be stored persistently, including the information of provisioned nodes, NetKey, AppKey, etc.

CONFIG_BLE_MESH_STORE_TIMEOUT

Delay (in seconds) before storing anything persistently

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_SETTINGS](#)

This value defines in seconds how soon any pending changes are actually written into persistent storage (flash) after a change occurs. The option allows nodes to delay a certain period of time to save proper information to flash. The default value is 0, which means information will be stored immediately once there are updates.

CONFIG_BLE_MESH_SEQ_STORE_RATE

How often the sequence number gets updated in storage

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_SETTINGS](#)

This value defines how often the local sequence number gets updated in persistent storage (i.e. flash). e.g. a value of 100 means that the sequence number will be stored to flash on every 100th increment. If the node sends messages very frequently a higher value makes more sense, whereas if the node sends infrequently a value as low as 0 (update storage for every increment) can make sense. When the stack gets initialized it will add sequence number to the last stored one, so that it starts off with a value that's guaranteed to be larger than the last one used before power off.

CONFIG_BLE_MESH_RPL_STORE_TIMEOUT

Minimum frequency that the RPL gets updated in storage

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_SETTINGS](#)

This value defines in seconds how soon the RPL (Replay Protection List) gets written to persistent storage after a change occurs. If the node receives messages frequently, then a large value is recommended. If the node receives messages rarely, then the value can be as low as 0 (which means the RPL is written into the storage immediately). Note that if the node operates in a security-sensitive case, and there is a risk of sudden power-off, then a value of 0 is strongly recommended. Otherwise, a power loss before RPL being written into the storage may introduce message replay attacks and system security will be in a vulnerable state.

CONFIG_BLE_MESH_SETTINGS_BACKWARD_COMPATIBILITY

A specific option for settings backward compatibility

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_SETTINGS](#)

This option is created to solve the issue of failure in recovering node information after mesh stack updates. In the old version mesh stack, there is no key of “mesh/role” in nvs. In the new version mesh stack, key of “mesh/role” is added in nvs, recovering node information needs to check “mesh/role” key in nvs and implements selective recovery of mesh node information. Therefore, there may be failure in recovering node information during node restarting after OTA.

The new version mesh stack adds the option of “mesh/role” because we have added the support of storing Provisioner information, while the old version only supports storing node information.

If users are updating their nodes from old version to new version, we recommend enabling this option, so that system could set the flag in advance before recovering node information and make sure the node information recovering could work as expected.

CONFIG_BLE_MESH_SPECIFIC_PARTITION

Use a specific NVS partition for BLE Mesh

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_SETTINGS](#)

When selected, the mesh stack will use a specified NVS partition instead of default NVS partition. Note that the specified partition must be registered with NVS using `nvs_flash_init_partition()` API, and the partition must exist in the csv file. When Provisioner needs to store a large amount of nodes' information in the flash (e.g. more than 20), this option is recommended to be enabled.

CONFIG_BLE_MESH_PARTITION_NAME

Name of the NVS partition for BLE Mesh

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_SETTINGS](#) > [CONFIG_BLE_MESH_SPECIFIC_PARTITION](#)

This value defines the name of the specified NVS partition used by the mesh stack.

CONFIG_BLE_MESH_USE_MULTIPLE_NAMESPACE

Support using multiple NVS namespaces by Provisioner

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_SETTINGS](#)

When selected, Provisioner can use different NVS namespaces to store different instances of mesh information. For example, if in the first room, Provisioner uses NetKey A, AppKey A and provisions three devices, these information will be treated as mesh information instance A. When the Provisioner moves to the second room, it uses NetKey B, AppKey B and provisions two devices, then the information will be treated as mesh information instance B. Here instance A and instance B will be stored in different namespaces. With this option enabled, Provisioner needs to use specific functions to open the corresponding NVS namespace, restore the mesh information, release the mesh information or erase the mesh information.

CONFIG_BLE_MESH_MAX_NVS_NAMESPACE

Maximum number of NVS namespaces

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_SETTINGS](#) > [CONFIG_BLE_MESH_USE_MULTIPLE_NAMESPACE](#)

This option specifies the maximum NVS namespaces supported by Provisioner.

CONFIG_BLE_MESH_SUBNET_COUNT

Maximum number of mesh subnets per network

Found in: [Component config](#) > [CONFIG_BLE_MESH](#)

This option specifies how many subnets a Mesh network can have at the same time. Indeed, this value decides the number of the network keys which can be owned by a node.

CONFIG_BLE_MESH_APP_KEY_COUNT

Maximum number of application keys per network

Found in: [Component config](#) > [CONFIG_BLE_MESH](#)

This option specifies how many application keys the device can store per network. Indeed, this value decides the number of the application keys which can be owned by a node.

CONFIG_BLE_MESH_MODEL_KEY_COUNT

Maximum number of application keys per model

Found in: [Component config](#) > [CONFIG_BLE_MESH](#)

This option specifies the maximum number of application keys to which each model can be bound.

CONFIG_BLE_MESH_MODEL_GROUP_COUNT

Maximum number of group address subscriptions per model

Found in: [Component config](#) > [CONFIG_BLE_MESH](#)

This option specifies the maximum number of addresses to which each model can be subscribed.

CONFIG_BLE_MESH_LABEL_COUNT

Maximum number of Label UUIDs used for Virtual Addresses

Found in: [Component config](#) > [CONFIG_BLE_MESH](#)

This option specifies how many Label UUIDs can be stored. Indeed, this value decides the number of the Virtual Addresses can be supported by a node.

CONFIG_BLE_MESH_CRPL

Maximum capacity of the replay protection list

Found in: [Component config](#) > [CONFIG_BLE_MESH](#)

This option specifies the maximum capacity of the replay protection list. It is similar to Network message cache size, but has a different purpose. The replay protection list is used to prevent a node from replay attack, which will store the source address and sequence number of the received mesh messages. For Provisioner, the replay protection list size should not be smaller than the maximum number of nodes whose information can be stored. And the element number of each node should also be taken into consideration. For example, if Provisioner can provision up to 20 nodes and each node contains two elements, then the replay protection list size of Provisioner should be at least 40.

CONFIG_BLE_MESH_MSG_CACHE_SIZE

Network message cache size

Found in: [Component config](#) > [CONFIG_BLE_MESH](#)

Number of messages that are cached for the network. This helps prevent unnecessary decryption operations and unnecessary relays. This option is similar to Replay protection list, but has a different purpose. A node is not required to cache the entire Network PDU and may cache only part of it for tracking, such as values for SRC/SEQ or others.

CONFIG_BLE_MESH_ADV_BUF_COUNT

Number of advertising buffers

Found in: [Component config](#) > [CONFIG_BLE_MESH](#)

Number of advertising buffers available. The transport layer reserves ADV_BUF_COUNT - 3 buffers for outgoing segments. The maximum outgoing SDU size is 12 times this value (out of which 4 or 8 bytes are used for the Transport Layer MIC). For example, 5 segments means the maximum SDU size is 60 bytes, which leaves 56 bytes for application layer data using a 4-byte MIC, or 52 bytes using an 8-byte MIC.

CONFIG_BLE_MESH_IVU_DIVIDER

Divider for IV Update state refresh timer

Found in: [Component config](#) > [CONFIG_BLE_MESH](#)

When the IV Update state enters Normal operation or IV Update in Progress, we need to keep track of how many hours has passed in the state, since the specification requires us to remain in the state at least for 96 hours (Update in Progress has an additional upper limit of 144 hours).

In order to fulfill the above requirement, even if the node might be powered off once in a while, we need to store persistently how many hours the node has been in the state. This doesn't necessarily need to happen every hour (thanks to the flexible duration range). The exact cadence will depend a lot on the ways that the node will be used and what kind of power source it has.

Since there is no single optimal answer, this configuration option allows specifying a divider, i.e. how many intervals the 96 hour minimum gets split into. After each interval the duration that the node has been in the current state gets stored to flash. E.g. the default value of 4 means that the state is saved every 24 hours (96 / 4).

CONFIG_BLE_MESH_TX_SEG_MSG_COUNT

Maximum number of simultaneous outgoing segmented messages

Found in: [Component config](#) > [CONFIG_BLE_MESH](#)

Maximum number of simultaneous outgoing multi-segment and/or reliable messages. The default value is 1, which means the device can only send one segmented message at a time. And if another segmented message is going to be sent, it should wait for the completion of the previous one. If users are going to send multiple segmented messages at the same time, this value should be configured properly.

CONFIG_BLE_MESH_RX_SEG_MSG_COUNT

Maximum number of simultaneous incoming segmented messages

Found in: [Component config](#) > [CONFIG_BLE_MESH](#)

Maximum number of simultaneous incoming multi-segment and/or reliable messages. The default value is 1, which means the device can only receive one segmented message at a time. And if another segmented message is going to be received, it should wait for the completion of the previous one. If users are going to receive multiple segmented messages at the same time, this value should be configured properly.

CONFIG_BLE_MESH_RX_SDU_MAX

Maximum incoming Upper Transport Access PDU length

Found in: [Component config](#) > [CONFIG_BLE_MESH](#)

Maximum incoming Upper Transport Access PDU length. Leave this to the default value, unless you really need to optimize memory usage.

CONFIG_BLE_MESH_TX_SEG_MAX

Maximum number of segments in outgoing messages

Found in: [Component config](#) > [CONFIG_BLE_MESH](#)

Maximum number of segments supported for outgoing messages. This value should typically be fine-tuned based on what models the local node supports, i.e. what's the largest message payload that the node needs to be able to send. This value affects memory and call stack consumption, which is why the default is lower than the maximum that the specification would allow (32 segments).

The maximum outgoing SDU size is 12 times this number (out of which 4 or 8 bytes is used for the Transport Layer MIC). For example, 5 segments means the maximum SDU size is 60 bytes, which leaves 56 bytes for application layer data using a 4-byte MIC and 52 bytes using an 8-byte MIC.

Be sure to specify a sufficient number of advertising buffers when setting this option to a higher value. There must be at least three more advertising buffers ([BLE_MESH_ADV_BUF_COUNT](#)) as there are outgoing segments.

CONFIG_BLE_MESH_RELAY

Relay support

Found in: [Component config](#) > [CONFIG_BLE_MESH](#)

Support for acting as a Mesh Relay Node. Enabling this option will allow a node to support the Relay feature, and the Relay feature can still be enabled or disabled by proper configuration messages. Disabling this option will let a node not support the Relay feature.

CONFIG_BLE_MESH_RELAY_ADV_BUF

Use separate advertising buffers for relay packets

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_RELAY](#)

When selected, self-send packets will be put in a high-priority queue and relay packets will be put in a low-priority queue.

CONFIG_BLE_MESH_RELAY_ADV_BUF_COUNT

Number of advertising buffers for relay packets

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_RELAY](#) > [CONFIG_BLE_MESH_RELAY_ADV_BUF](#)

Number of advertising buffers for relay packets available.

CONFIG_BLE_MESH_LOW_POWER

Support for Low Power features

Found in: [Component config](#) > [CONFIG_BLE_MESH](#)

Enable this option to operate as a Low Power Node. If low power consumption is required by a node, this option should be enabled. And once the node enters the mesh network, it will try to find a Friend node and establish a friendship.

CONFIG_BLE_MESH_LPN_ESTABLISHMENT

Perform Friendship establishment using low power

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_LOW_POWER](#)

Perform the Friendship establishment using low power with the help of a reduced scan duty cycle. The downside of this is that the node may miss out on messages intended for it until it has successfully set up

Friendship with a Friend node. When this option is enabled, the node will stop scanning for a period of time after a Friend Request or Friend Poll is sent, so as to reduce more power consumption.

CONFIG_BLE_MESH_LPN_AUTO

Automatically start looking for Friend nodes once provisioned

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_LOW_POWER](#)

Once provisioned, automatically enable LPN functionality and start looking for Friend nodes. If this option is disabled LPN mode needs to be manually enabled by calling `bt_mesh_lpn_set(true)`. When an unprovisioned device is provisioned successfully and becomes a node, enabling this option will trigger the node starts to send Friend Request at a certain period until it finds a proper Friend node.

CONFIG_BLE_MESH_LPN_AUTO_TIMEOUT

Time from last received message before going to LPN mode

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_LOW_POWER](#) > [CONFIG_BLE_MESH_LPN_AUTO](#)

Time in seconds from the last received message, that the node waits out before starting to look for Friend nodes.

CONFIG_BLE_MESH_LPN_RETRY_TIMEOUT

Retry timeout for Friend requests

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_LOW_POWER](#)

Time in seconds between Friend Requests, if a previous Friend Request did not yield any acceptable Friend Offers.

CONFIG_BLE_MESH_LPN_RSSI_FACTOR

RSSIFactor, used in Friend Offer Delay calculation

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_LOW_POWER](#)

The contribution of the RSSI, measured by the Friend node, used in Friend Offer Delay calculations. 0 = 1, 1 = 1.5, 2 = 2, 3 = 2.5. RSSIFactor, one of the parameters carried by Friend Request sent by Low Power node, which is used to calculate the Friend Offer Delay.

CONFIG_BLE_MESH_LPN_RECV_WIN_FACTOR

ReceiveWindowFactor, used in Friend Offer Delay calculation

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_LOW_POWER](#)

The contribution of the supported Receive Window used in Friend Offer Delay calculations. 0 = 1, 1 = 1.5, 2 = 2, 3 = 2.5. ReceiveWindowFactor, one of the parameters carried by Friend Request sent by Low Power node, which is used to calculate the Friend Offer Delay.

CONFIG_BLE_MESH_LPN_MIN_QUEUE_SIZE

Minimum size of the acceptable friend queue (MinQueueSizeLog)

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_LOW_POWER](#)

The MinQueueSizeLog field is defined as $\log_2(N)$, where N is the minimum number of maximum size Lower Transport PDUs that the Friend node can store in its Friend Queue. As an example, MinQueueSizeLog value 1 gives $N = 2$, and value 7 gives $N = 128$.

CONFIG_BLE_MESH_LPN_RECV_DELAY

Receive delay requested by the local node

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_LOW_POWER](#)

The ReceiveDelay is the time between the Low Power node sending a request and listening for a response. This delay allows the Friend node time to prepare the response. The value is in units of milliseconds.

CONFIG_BLE_MESH_LPN_POLL_TIMEOUT

The value of the PollTimeout timer

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_LOW_POWER](#)

PollTimeout timer is used to measure time between two consecutive requests sent by a Low Power node. If no requests are received the Friend node before the PollTimeout timer expires, then the friendship is considered terminated. The value is in units of 100 milliseconds, so e.g. a value of 300 means 30 seconds. The smaller the value, the faster the Low Power node tries to get messages from corresponding Friend node and vice versa.

CONFIG_BLE_MESH_LPN_INIT_POLL_TIMEOUT

The starting value of the PollTimeout timer

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_LOW_POWER](#)

The initial value of the PollTimeout timer when Friendship is to be established for the first time. After this, the timeout gradually grows toward the actual PollTimeout, doubling in value for each iteration. The value is in units of 100 milliseconds, so e.g. a value of 300 means 30 seconds.

CONFIG_BLE_MESH_LPN_SCAN_LATENCY

Latency for enabling scanning

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_LOW_POWER](#)

Latency (in milliseconds) is the time it takes to enable scanning. In practice, it means how much time in advance of the Receive Window, the request to enable scanning is made.

CONFIG_BLE_MESH_LPN_GROUPS

Number of groups the LPN can subscribe to

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_LOW_POWER](#)

Maximum number of groups to which the LPN can subscribe.

CONFIG_BLE_MESH_FRIEND

Support for Friend feature

Found in: [Component config](#) > [CONFIG_BLE_MESH](#)

Enable this option to be able to act as a Friend Node.

CONFIG_BLE_MESH_FRIEND_RECV_WIN

Friend Receive Window

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_FRIEND](#)

Receive Window in milliseconds supported by the Friend node.

CONFIG_BLE_MESH_FRIEND_QUEUE_SIZE

Minimum number of buffers supported per Friend Queue

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_FRIEND](#)

Minimum number of buffers available to be stored for each local Friend Queue. This option decides the size of each buffer which can be used by a Friend node to store messages for each Low Power node.

CONFIG_BLE_MESH_FRIEND_SUB_LIST_SIZE

Friend Subscription List Size

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_FRIEND](#)

Size of the Subscription List that can be supported by a Friend node for a Low Power node. And Low Power node can send Friend Subscription List Add or Friend Subscription List Remove messages to the Friend node to add or remove subscription addresses.

CONFIG_BLE_MESH_FRIEND_LPN_COUNT

Number of supported LPN nodes

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_FRIEND](#)

Number of Low Power Nodes with which a Friend can have Friendship simultaneously. A Friend node can have friendship with multiple Low Power nodes at the same time, while a Low Power node can only establish friendship with only one Friend node at the same time.

CONFIG_BLE_MESH_FRIEND_SEG_RX

Number of incomplete segment lists per LPN

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [CONFIG_BLE_MESH_FRIEND](#)

Number of incomplete segment lists tracked for each Friends' LPN. In other words, this determines from how many elements can segmented messages destined for the Friend queue be received simultaneously.

CONFIG_BLE_MESH_NO_LOG

Disable BLE Mesh debug logs (minimize bin size)

Found in: [Component config](#) > [CONFIG_BLE_MESH](#)

Select this to save the BLE Mesh related rodata code size. Enabling this option will disable the output of BLE Mesh debug log.

BLE Mesh STACK DEBUG LOG LEVEL Contains:

- [CONFIG_BLE_MESH_STACK_TRACE_LEVEL](#)

CONFIG_BLE_MESH_STACK_TRACE_LEVEL

BLE_MESH_STACK

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [BLE Mesh STACK DEBUG LOG LEVEL](#)

Define BLE Mesh trace level for BLE Mesh stack.

Available options:

- NONE (BLE_MESH_TRACE_LEVEL_NONE)
- ERROR (BLE_MESH_TRACE_LEVEL_ERROR)
- WARNING (BLE_MESH_TRACE_LEVEL_WARNING)
- INFO (BLE_MESH_TRACE_LEVEL_INFO)

- DEBUG (BLE_MESH_TRACE_LEVEL_DEBUG)
- VERBOSE (BLE_MESH_TRACE_LEVEL_VERBOSE)

BLE Mesh NET BUF DEBUG LOG LEVEL Contains:

- [CONFIG_BLE_MESH_NET_BUF_TRACE_LEVEL](#)

CONFIG_BLE_MESH_NET_BUF_TRACE_LEVEL

BLE_MESH_NET_BUF

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [BLE Mesh NET BUF DEBUG LOG LEVEL](#)

Define BLE Mesh trace level for BLE Mesh net buffer.

Available options:

- NONE (BLE_MESH_NET_BUF_TRACE_LEVEL_NONE)
- ERROR (BLE_MESH_NET_BUF_TRACE_LEVEL_ERROR)
- WARNING (BLE_MESH_NET_BUF_TRACE_LEVEL_WARNING)
- INFO (BLE_MESH_NET_BUF_TRACE_LEVEL_INFO)
- DEBUG (BLE_MESH_NET_BUF_TRACE_LEVEL_DEBUG)
- VERBOSE (BLE_MESH_NET_BUF_TRACE_LEVEL_VERBOSE)

CONFIG_BLE_MESH_CLIENT_MSG_TIMEOUT

Timeout(ms) for client message response

Found in: [Component config](#) > [CONFIG_BLE_MESH](#)

Timeout value used by the node to get response of the acknowledged message which is sent by the client model. This value indicates the maximum time that a client model waits for the response of the sent acknowledged messages. If a client model uses 0 as the timeout value when sending acknowledged messages, then the default value will be used which is four seconds.

Support for BLE Mesh Foundation models Contains:

- [CONFIG_BLE_MESH_CFG_CLI](#)
- [CONFIG_BLE_MESH_HEALTH_CLI](#)
- [CONFIG_BLE_MESH_HEALTH_SRV](#)

CONFIG_BLE_MESH_CFG_CLI

Configuration Client model

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [Support for BLE Mesh Foundation models](#)

Enable support for Configuration Client model.

CONFIG_BLE_MESH_HEALTH_CLI

Health Client model

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [Support for BLE Mesh Foundation models](#)

Enable support for Health Client model.

CONFIG_BLE_MESH_HEALTH_SRV

Health Server model

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models

Enable support for Health Server model.

Support for BLE Mesh Client/Server models Contains:

- *CONFIG_BLE_MESH_GENERIC_ONOFF_CLI*
- *CONFIG_BLE_MESH_GENERIC_LEVEL_CLI*
- *CONFIG_BLE_MESH_GENERIC_DEF_TRANS_TIME_CLI*
- *CONFIG_BLE_MESH_GENERIC_POWER_ONOFF_CLI*
- *CONFIG_BLE_MESH_GENERIC_POWER_LEVEL_CLI*
- *CONFIG_BLE_MESH_GENERIC_BATTERY_CLI*
- *CONFIG_BLE_MESH_GENERIC_LOCATION_CLI*
- *CONFIG_BLE_MESH_GENERIC_PROPERTY_CLI*
- *CONFIG_BLE_MESH_SENSOR_CLI*
- *CONFIG_BLE_MESH_TIME_CLI*
- *CONFIG_BLE_MESH_SCENE_CLI*
- *CONFIG_BLE_MESH_SCHEDULER_CLI*
- *CONFIG_BLE_MESH_LIGHT_LIGHTNESS_CLI*
- *CONFIG_BLE_MESH_LIGHT_CTL_CLI*
- *CONFIG_BLE_MESH_LIGHT_HSL_CLI*
- *CONFIG_BLE_MESH_LIGHT_XYL_CLI*
- *CONFIG_BLE_MESH_LIGHT_LC_CLI*
- *CONFIG_BLE_MESH_GENERIC_SERVER*
- *CONFIG_BLE_MESH_SENSOR_SERVER*
- *CONFIG_BLE_MESH_TIME_SCENE_SERVER*
- *CONFIG_BLE_MESH_LIGHTING_SERVER*

CONFIG_BLE_MESH_GENERIC_ONOFF_CLI

Generic OnOff Client model

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Client/Server models

Enable support for Generic OnOff Client model.

CONFIG_BLE_MESH_GENERIC_LEVEL_CLI

Generic Level Client model

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Client/Server models

Enable support for Generic Level Client model.

CONFIG_BLE_MESH_GENERIC_DEF_TRANS_TIME_CLI

Generic Default Transition Time Client model

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Client/Server models

Enable support for Generic Default Transition Time Client model.

CONFIG_BLE_MESH_GENERIC_POWER_ONOFF_CLI

Generic Power OnOff Client model

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Client/Server models

Enable support for Generic Power OnOff Client model.

CONFIG_BLE_MESH_GENERIC_POWER_LEVEL_CLI

Generic Power Level Client model

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [Support for BLE Mesh Client/Server models](#)

Enable support for Generic Power Level Client model.

CONFIG_BLE_MESH_GENERIC_BATTERY_CLI

Generic Battery Client model

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [Support for BLE Mesh Client/Server models](#)

Enable support for Generic Battery Client model.

CONFIG_BLE_MESH_GENERIC_LOCATION_CLI

Generic Location Client model

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [Support for BLE Mesh Client/Server models](#)

Enable support for Generic Location Client model.

CONFIG_BLE_MESH_GENERIC_PROPERTY_CLI

Generic Property Client model

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [Support for BLE Mesh Client/Server models](#)

Enable support for Generic Property Client model.

CONFIG_BLE_MESH_SENSOR_CLI

Sensor Client model

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [Support for BLE Mesh Client/Server models](#)

Enable support for Sensor Client model.

CONFIG_BLE_MESH_TIME_CLI

Time Client model

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [Support for BLE Mesh Client/Server models](#)

Enable support for Time Client model.

CONFIG_BLE_MESH_SCENE_CLI

Scene Client model

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [Support for BLE Mesh Client/Server models](#)

Enable support for Scene Client model.

CONFIG_BLE_MESH_SCHEDULER_CLI

Scheduler Client model

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [Support for BLE Mesh Client/Server models](#)

Enable support for Scheduler Client model.

CONFIG_BLE_MESH_LIGHT_LIGHTNESS_CLI

Light Lightness Client model

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [Support for BLE Mesh Client/Server models](#)

Enable support for Light Lightness Client model.

CONFIG_BLE_MESH_LIGHT_CTL_CLI

Light CTL Client model

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [Support for BLE Mesh Client/Server models](#)

Enable support for Light CTL Client model.

CONFIG_BLE_MESH_LIGHT_HSL_CLI

Light HSL Client model

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [Support for BLE Mesh Client/Server models](#)

Enable support for Light HSL Client model.

CONFIG_BLE_MESH_LIGHT_XYL_CLI

Light XYL Client model

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [Support for BLE Mesh Client/Server models](#)

Enable support for Light XYL Client model.

CONFIG_BLE_MESH_LIGHT_LC_CLI

Light LC Client model

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [Support for BLE Mesh Client/Server models](#)

Enable support for Light LC Client model.

CONFIG_BLE_MESH_GENERIC_SERVER

Generic server models

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [Support for BLE Mesh Client/Server models](#)

Enable support for Generic server models.

CONFIG_BLE_MESH_SENSOR_SERVER

Sensor server models

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [Support for BLE Mesh Client/Server models](#)

Enable support for Sensor server models.

CONFIG_BLE_MESH_TIME_SCENE_SERVER

Time and Scenes server models

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [Support for BLE Mesh Client/Server models](#)

Enable support for Time and Scenes server models.

CONFIG_BLE_MESH_LIGHTING_SERVER

Lighting server models

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Client/Server models

Enable support for Lighting server models.

CONFIG_BLE_MESH_IV_UPDATE_TEST

Test the IV Update Procedure

Found in: Component config > CONFIG_BLE_MESH

This option removes the 96 hour limit of the IV Update Procedure and lets the state to be changed at any time. If IV Update test mode is going to be used, this option should be enabled.

BLE Mesh specific test option Contains:

- *CONFIG_BLE_MESH_SELF_TEST*
- *CONFIG_BLE_MESH_SHELL*
- *CONFIG_BLE_MESH_DEBUG*

CONFIG_BLE_MESH_SELF_TEST

Perform BLE Mesh self-tests

Found in: Component config > CONFIG_BLE_MESH > BLE Mesh specific test option

This option adds extra self-tests which are run every time BLE Mesh networking is initialized.

CONFIG_BLE_MESH_TEST_AUTO_ENTER_NETWORK

Unprovisioned device enters mesh network automatically

Found in: Component config > CONFIG_BLE_MESH > BLE Mesh specific test option > CONFIG_BLE_MESH_SELF_TEST

With this option enabled, an unprovisioned device can automatically enters mesh network using a specific test function without the provisioning procedure. And on the Provisioner side, a test function needs to be invoked to add the node information into the mesh stack.

CONFIG_BLE_MESH_TEST_USE_WHITE_LIST

Use white list to filter mesh advertising packets

Found in: Component config > CONFIG_BLE_MESH > BLE Mesh specific test option > CONFIG_BLE_MESH_SELF_TEST

With this option enabled, users can use white list to filter mesh advertising packets while scanning.

CONFIG_BLE_MESH_SHELL

Enable BLE Mesh shell

Found in: Component config > CONFIG_BLE_MESH > BLE Mesh specific test option

Activate shell module that provides BLE Mesh commands to the console.

CONFIG_BLE_MESH_DEBUG

Enable BLE Mesh debug logs

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [BLE Mesh specific test option](#)

Enable debug logs for the BLE Mesh functionality.

CONFIG_BLE_MESH_DEBUG_NET

Network layer debug

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [BLE Mesh specific test option](#) > [CONFIG_BLE_MESH_DEBUG](#)

Enable Network layer debug logs for the BLE Mesh functionality.

CONFIG_BLE_MESH_DEBUG_TRANS

Transport layer debug

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [BLE Mesh specific test option](#) > [CONFIG_BLE_MESH_DEBUG](#)

Enable Transport layer debug logs for the BLE Mesh functionality.

CONFIG_BLE_MESH_DEBUG_BEACON

Beacon debug

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [BLE Mesh specific test option](#) > [CONFIG_BLE_MESH_DEBUG](#)

Enable Beacon-related debug logs for the BLE Mesh functionality.

CONFIG_BLE_MESH_DEBUG_CRYPTO

Crypto debug

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [BLE Mesh specific test option](#) > [CONFIG_BLE_MESH_DEBUG](#)

Enable cryptographic debug logs for the BLE Mesh functionality.

CONFIG_BLE_MESH_DEBUG_PROV

Provisioning debug

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [BLE Mesh specific test option](#) > [CONFIG_BLE_MESH_DEBUG](#)

Enable Provisioning debug logs for the BLE Mesh functionality.

CONFIG_BLE_MESH_DEBUG_ACCESS

Access layer debug

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [BLE Mesh specific test option](#) > [CONFIG_BLE_MESH_DEBUG](#)

Enable Access layer debug logs for the BLE Mesh functionality.

CONFIG_BLE_MESH_DEBUG_MODEL

Foundation model debug

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [BLE Mesh specific test option](#) > [CONFIG_BLE_MESH_DEBUG](#)

Enable Foundation Models debug logs for the BLE Mesh functionality.

CONFIG_BLE_MESH_DEBUG_ADV

Advertising debug

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [BLE Mesh specific test option](#) > [CONFIG_BLE_MESH_DEBUG](#)

Enable advertising debug logs for the BLE Mesh functionality.

CONFIG_BLE_MESH_DEBUG_LOW_POWER

Low Power debug

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [BLE Mesh specific test option](#) > [CONFIG_BLE_MESH_DEBUG](#)

Enable Low Power debug logs for the BLE Mesh functionality.

CONFIG_BLE_MESH_DEBUG_FRIEND

Friend debug

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [BLE Mesh specific test option](#) > [CONFIG_BLE_MESH_DEBUG](#)

Enable Friend debug logs for the BLE Mesh functionality.

CONFIG_BLE_MESH_DEBUG_PROXY

Proxy debug

Found in: [Component config](#) > [CONFIG_BLE_MESH](#) > [BLE Mesh specific test option](#) > [CONFIG_BLE_MESH_DEBUG](#)

Enable Proxy protocol debug logs for the BLE Mesh functionality.

CoAP Configuration Contains:

- [CONFIG_COAP_MBEDTLS_ENCRYPTION_MODE](#)
- [CONFIG_COAP_MBEDTLS_DEBUG](#)

CONFIG_COAP_MBEDTLS_ENCRYPTION_MODE

CoAP Encryption method

Found in: [Component config](#) > [CoAP Configuration](#)

If the CoAP information is to be encrypted, the encryption environment can be set up in one of two ways (default being Pre-Shared key mode)

- Encrypt using defined Pre-Shared Keys (PSK if uri includes coaps://)
- Encrypt using defined Public Key Infrastructure (PKI if uri includes coaps://)

Available options:

- Pre-Shared Keys (COAP_MBEDTLS_PSK)
- PKI Certificates (COAP_MBEDTLS_PKI)

CONFIG_COAP_MBEDTLS_DEBUG

Enable CoAP debugging

Found in: [Component config](#) > [CoAP Configuration](#)

Enable CoAP debugging functions at compile time for the example code.

If this option is enabled, call `coap_set_log_level()` at runtime in order to enable CoAP debug output via the ESP log mechanism.

CONFIG_COAP_MBEDTLS_DEBUG_LEVEL

Set CoAP debugging level

Found in: [Component config](#) > [CoAP Configuration](#) > [CONFIG_COAP_MBEDTLS_DEBUG](#)

Set CoAP debugging level

Available options:

- Emergency (COAP_LOG_EMERG)
- Alert (COAP_LOG_ALERT)
- Critical (COAP_LOG_CRIT)
- Error (COAP_LOG_ERROR)
- Warning (COAP_LOG_WARNING)
- Notice (COAP_LOG_NOTICE)
- Info (COAP_LOG_INFO)
- Debug (COAP_LOG_DEBUG)

Driver configurations

 Contains:

- [ADC configuration](#)
- [SPI configuration](#)
- [CAN configuration](#)
- [UART configuration](#)
- [RTCIO configuration](#)

ADC configuration

 Contains:

- [CONFIG_ADC_FORCE_XPD_FSM](#)
- [CONFIG_ADC_DISABLE_DAC](#)

CONFIG_ADC_FORCE_XPD_FSM

Use the FSM to control ADC power

Found in: [Component config](#) > [Driver configurations](#) > [ADC configuration](#)

ADC power can be controlled by the FSM instead of software. This allows the ADC to be shut off when it is not working leading to lower power consumption. However using the FSM control ADC power will increase the noise of ADC.

CONFIG_ADC_DISABLE_DAC

Disable DAC when ADC2 is used on GPIO 25 and 26

Found in: [Component config](#) > [Driver configurations](#) > [ADC configuration](#)

If this is set, the ADC2 driver will disable the output of the DAC corresponding to the specified channel. This is the default value.

For testing, disable this option so that we can measure the output of DAC by internal ADC.

SPI configuration Contains:

- [CONFIG_SPI_MASTER_IN_IRAM](#)
- [CONFIG_SPI_MASTER_ISR_IN_IRAM](#)
- [CONFIG_SPI_SLAVE_IN_IRAM](#)
- [CONFIG_SPI_SLAVE_ISR_IN_IRAM](#)

CONFIG_SPI_MASTER_IN_IRAM

Place transmitting functions of SPI master into IRAM

Found in: [Component config](#) > [Driver configurations](#) > [SPI configuration](#)

Normally only the ISR of SPI master is placed in the IRAM, so that it can work without the flash when interrupt is triggered. For other functions, there's some possibility that the flash cache miss when running inside and out of SPI functions, which may increase the interval of SPI transactions. Enable this to put `queue_trans`, `get_trans_result` and `transmit` functions into the IRAM to avoid possible cache miss.

During unit test, this is enabled to measure the ideal case of api.

CONFIG_SPI_MASTER_ISR_IN_IRAM

Place SPI master ISR function into IRAM

Found in: [Component config](#) > [Driver configurations](#) > [SPI configuration](#)

Place the SPI master ISR in to IRAM to avoid possible cache miss.

Also you can forbid the ISR being disabled during flash writing access, by add `ESP_INTR_FLAG_IRAM` when initializing the driver.

CONFIG_SPI_SLAVE_IN_IRAM

Place transmitting functions of SPI slave into IRAM

Found in: [Component config](#) > [Driver configurations](#) > [SPI configuration](#)

Normally only the ISR of SPI slave is placed in the IRAM, so that it can work without the flash when interrupt is triggered. For other functions, there's some possibility that the flash cache miss when running inside and out of SPI functions, which may increase the interval of SPI transactions. Enable this to put `queue_trans`, `get_trans_result` and `transmit` functions into the IRAM to avoid possible cache miss.

CONFIG_SPI_SLAVE_ISR_IN_IRAM

Place SPI slave ISR function into IRAM

Found in: [Component config](#) > [Driver configurations](#) > [SPI configuration](#)

Place the SPI slave ISR in to IRAM to avoid possible cache miss.

Also you can forbid the ISR being disabled during flash writing access, by add `ESP_INTR_FLAG_IRAM` when initializing the driver.

CAN configuration Contains:

- [CONFIG_CAN_ISR_IN_IRAM](#)

CONFIG_CAN_ISR_IN_IRAM

Place CAN ISR function into IRAM

Found in: [Component config](#) > [Driver configurations](#) > [CAN configuration](#)

Place the CAN ISR in to IRAM. This will allow the ISR to avoid cache misses, and also be able to run whilst the cache is disabled (such as when writing to SPI Flash). Note that if this option is enabled: - Users should also set the ESP_INTR_FLAG_IRAM in the driver configuration structure when installing the driver (see docs for specifics). - Alert logging (i.e., setting of the CAN_ALERT_AND_LOG flag) will have no effect.

UART configuration Contains:

- [CONFIG_UART_ISR_IN_IRAM](#)

CONFIG_UART_ISR_IN_IRAM

Place UART ISR function into IRAM

Found in: [Component config](#) > [Driver configurations](#) > [UART configuration](#)

If this option is not selected, UART interrupt will be disabled for a long time and may cause data lost when doing spi flash operation.

RTCIO configuration Contains:

- [CONFIG_RTCIO_SUPPORT_RTC_GPIO_DESC](#)

CONFIG_RTCIO_SUPPORT_RTC_GPIO_DESC

Support array *rtc_gpio_desc* for ESP32

Found in: [Component config](#) > [Driver configurations](#) > [RTCIO configuration](#)

The the array *rtc_gpio_desc* will don' t compile by default. If this option is selected, the array *rtc_gpio_desc* can be compile. If user use this array, please enable this configuration.

eFuse Bit Manager Contains:

- [CONFIG_EFUSE_CUSTOM_TABLE](#)
- [CONFIG_EFUSE_VIRTUAL](#)
- [CONFIG_EFUSE_CODE_SCHEME_SELECTOR](#)

CONFIG_EFUSE_CUSTOM_TABLE

Use custom eFuse table

Found in: [Component config](#) > [eFuse Bit Manager](#)

Allows to generate a structure for eFuse from the CSV file.

CONFIG_EFUSE_CUSTOM_TABLE_FILENAME

Custom eFuse CSV file

Found in: [Component config](#) > [eFuse Bit Manager](#) > [CONFIG_EFUSE_CUSTOM_TABLE](#)

Name of the custom eFuse CSV filename. This path is evaluated relative to the project root directory.

CONFIG_EFUSE_VIRTUAL

Simulate eFuse operations in RAM

Found in: [Component config](#) > [eFuse Bit Manager](#)

All read and writes operations are redirected to RAM instead of eFuse registers. If this option is set, all permanent changes (via eFuse) are disabled. Log output will state changes which would be applied, but they will not be.

CONFIG_EFUSE_CODE_SCHEME_SELECTOR

Coding Scheme Compatibility

Found in: [Component config](#) > [eFuse Bit Manager](#)

Selector eFuse code scheme.

Available options:

- None Only (EFUSE_CODE_SCHEME_COMPAT_NONE)
- 3/4 and None (EFUSE_CODE_SCHEME_COMPAT_3_4)
- Repeat, 3/4 and None (common table does not support it) (EFUSE_CODE_SCHEME_COMPAT_REPEAT)

ESP-TLS

 Contains:

- [CONFIG_ESP_TLS_LIBRARY_CHOOSE](#)
- [CONFIG_ESP_TLS_SERVER](#)
- [CONFIG_ESP_TLS_PSK_VERIFICATION](#)
- [CONFIG_ESP_WOLFSSL_SMALL_CERT_VERIFY](#)
- [CONFIG_ESP_DEBUG_WOLFSSL](#)

CONFIG_ESP_TLS_LIBRARY_CHOOSE

Choose SSL/TLS library for ESP-TLS (See help for more Info)

Found in: [Component config](#) > [ESP-TLS](#)

The ESP-TLS APIs support multiple backend TLS libraries. Currently mbedTLS and WolfSSL are supported. Different TLS libraries may support different features and have different resource usage. Consult the ESP-TLS documentation in ESP-IDF Programming guide for more details.

Available options:

- mbedTLS (ESP_TLS_USING_MBEDTLS)
- wolfSSL (License info in wolfSSL directory README) (ESP_TLS_USING_WOLFSSL)

CONFIG_ESP_TLS_SERVER

Enable ESP-TLS Server

Found in: [Component config](#) > [ESP-TLS](#)

Enable support for creating server side SSL/TLS session, uses the mbedtls crypto library

CONFIG_ESP_TLS_PSK_VERIFICATION

Enable PSK verification

Found in: [Component config](#) > [ESP-TLS](#)

Enable support for pre shared key ciphers, uses the mbedtls crypto library

CONFIG_ESP_WOLFSSL_SMALL_CERT_VERIFY

Enable SMALL_CERT_VERIFY

Found in: [Component config](#) > [ESP-TLS](#)

Enables server verification with Intermediate CA cert, does not authenticate full chain of trust up to the root CA cert (After Enabling this option client only needs to have Intermediate CA certificate of the server to authenticate server, root CA cert is not necessary).

CONFIG_ESP_DEBUG_WOLFSSL

Enable debug logs for wolfSSL

Found in: [Component config](#) > [ESP-TLS](#)

Enable detailed debug prints for wolfSSL SSL library.

ESP32-specific

 Contains:

- [CONFIG_ESP32_REV_MIN](#)
- [CONFIG_ESP32_DEFAULT_CPU_FREQ_MHZ](#)
- [CONFIG_ESP32_SPIRAM_SUPPORT](#)
- [CONFIG_ESP32_TRAX](#)
- [CONFIG_ESP32_UNIVERSAL_MAC_ADDRESSES](#)
- [CONFIG_ESP32_ULP_COPROC_ENABLED](#)
- [CONFIG_ESP32_PANIC](#)
- [CONFIG_ESP32_DEBUG_OCDAWARE](#)
- [CONFIG_ESP32_BROWNOUT_DET](#)
- [CONFIG_ESP32_REDUCE_PHY_TX_POWER](#)
- [CONFIG_ESP32_TIME_SYSCALL](#)
- [CONFIG_ESP32_RTC_CLK_SRC](#)
- [CONFIG_ESP32_RTC_EXT_CRYST_ADDIT_CURRENT](#)
- [CONFIG_ESP32_RTC_CLK_CAL_CYCLES](#)
- [CONFIG_ESP32_RTC_XTAL_CAL_RETRY](#)
- [CONFIG_ESP32_RTC_XTAL_BOOTSTRAP_CYCLES](#)
- [CONFIG_ESP32_DEEP_SLEEP_WAKEUP_DELAY](#)
- [CONFIG_ESP32_XTAL_FREQ_SEL](#)
- [CONFIG_ESP32_DISABLE_BASIC_ROM_CONSOLE](#)
- [CONFIG_ESP32_NO_BLOBS](#)
- [CONFIG_ESP32_COMPATIBLE_PRE_V2_1_BOOTLOADERS](#)
- [CONFIG_ESP32_COMPATIBLE_PRE_V3_1_BOOTLOADERS](#)
- [CONFIG_ESP32_RTCDATA_IN_FAST_MEM](#)
- [CONFIG_ESP32_USE_FIXED_STATIC_RAM_SIZE](#)
- [CONFIG_ESP32_DPORT_DIS_INTERRUPT_LVL](#)

CONFIG_ESP32_REV_MIN

Minimum Supported ESP32 Revision

Found in: [Component config](#) > [ESP32-specific](#)

Minimum revision that ESP-IDF would support. ESP-IDF performs different strategy on different esp32 revision.

Available options:

- Rev 0 ([ESP32_REV_MIN_0](#))
- Rev 1 ([ESP32_REV_MIN_1](#))
- Rev 2 ([ESP32_REV_MIN_2](#))
- Rev 3 ([ESP32_REV_MIN_3](#))

CONFIG_ESP32_DEFAULT_CPU_FREQ_MHZ

CPU frequency

Found in: *Component config > ESP32-specific*

CPU frequency to be set on application startup.

Available options:

- 80 MHz (ESP32_DEFAULT_CPU_FREQ_80)
- 160 MHz (ESP32_DEFAULT_CPU_FREQ_160)
- 240 MHz (ESP32_DEFAULT_CPU_FREQ_240)

CONFIG_ESP32_SPIRAM_SUPPORT

Support for external, SPI-connected RAM

Found in: *Component config > ESP32-specific*

This enables support for an external SPI RAM chip, connected in parallel with the main SPI flash chip.

SPI RAM config

 Contains:

- *CONFIG_SPIRAM_TYPE*
- *CONFIG_SPIRAM_SPEED*
- *CONFIG_SPIRAM_BOOT_INIT*
- *CONFIG_SPIRAM_USE*
- *CONFIG_SPIRAM_MEMTEST*
- *CONFIG_SPIRAM_MALLOC_ALWAYSINTERNAL*
- *CONFIG_SPIRAM_TRY_ALLOCATE_WIFI_LWIP*
- *CONFIG_SPIRAM_MALLOC_RESERVE_INTERNAL*
- *CONFIG_SPIRAM_ALLOW_BSS_SEG_EXTERNAL_MEMORY*
- *CONFIG_SPIRAM_CACHE_WORKAROUND*
- *SPIRAM cache workaround debugging*
- *CONFIG_SPIRAM_BANKSWITCH_ENABLE*
- *CONFIG_SPIRAM_ALLOW_STACK_EXTERNAL_MEMORY*
- *CONFIG_SPIRAM_OCCUPY_SPI_HOST*
- *PSRAM clock and cs IO for ESP32-D0WD*
- *PSRAM clock and cs IO for ESP32-D2WD*
- *PSRAM clock and cs IO for ESP32-PICO*
- *CONFIG_SPIRAM_CUSTOM_SPIWP_SD3_PIN*
- *CONFIG_SPIRAM_SPIWP_SD3_PIN*
- *CONFIG_SPIRAM_2T_MODE*

CONFIG_SPIRAM_TYPE

Type of SPI RAM chip in use

Found in: *Component config > ESP32-specific > CONFIG_ESP32_SPIRAM_SUPPORT > SPI RAM config*

Available options:

- Auto-detect (SPIRAM_TYPE_AUTO)
- ESP-PSRAM16 or APS1604 (SPIRAM_TYPE_ESPPSRAM16)
- ESP-PSRAM32 or IS25WP032 (SPIRAM_TYPE_ESPPSRAM32)
- ESP-PSRAM64 or LY68L6400 (SPIRAM_TYPE_ESPPSRAM64)

CONFIG_SPIRAM_SPEED

Set RAM clock speed

Found in: *Component config > ESP32-specific > CONFIG_ESP32_SPIRAM_SUPPORT > SPI RAM config*

Select the speed for the SPI RAM chip. If SPI RAM is enabled, we only support three combinations of SPI speed mode we supported now:

1. Flash SPI running at 40Mhz and RAM SPI running at 40Mhz
2. Flash SPI running at 80Mhz and RAM SPI running at 40Mhz
3. Flash SPI running at 80Mhz and RAM SPI running at 80Mhz

Note: If the third mode(80Mhz+80Mhz) is enabled for SPI RAM of type 32MBit, one of the HSPI/VSPI host will be occupied by the system. Which SPI host to use can be selected by the config item SPIRAM_OCCUPY_SPI_HOST. Application code should never touch HSPI/VSPI hardware in this case. The option to select 80MHz will only be visible if the flash SPI speed is also 80MHz. (ESP_TOOLPY_FLASHFREQ_80M is true)

Available options:

- 40MHz clock speed (SPIRAM_SPEED_40M)
- 80MHz clock speed (SPIRAM_SPEED_80M)

CONFIG_SPIRAM_BOOT_INIT

Initialize SPI RAM during startup

Found in: [Component config](#) > [ESP32-specific](#) > [CONFIG_ESP32_SPIRAM_SUPPORT](#) > [SPI RAM config](#)

If this is enabled, the SPI RAM will be enabled during initial boot. Unless you have specific requirements, you'll want to leave this enabled so memory allocated during boot-up can also be placed in SPI RAM.

CONFIG_SPIRAM_IGNORE_NOTFOUND

Ignore PSRAM when not found

Found in: [Component config](#) > [ESP32-specific](#) > [CONFIG_ESP32_SPIRAM_SUPPORT](#) > [SPI RAM config](#) > [CONFIG_SPIRAM_BOOT_INIT](#)

Normally, if psram initialization is enabled during compile time but not found at runtime, it is seen as an error making the CPU panic. If this is enabled, booting will complete but no PSRAM will be available.

CONFIG_SPIRAM_USE

SPI RAM access method

Found in: [Component config](#) > [ESP32-specific](#) > [CONFIG_ESP32_SPIRAM_SUPPORT](#) > [SPI RAM config](#)

The SPI RAM can be accessed in multiple methods: by just having it available as an unmanaged memory region in the CPU's memory map, by integrating it in the heap as 'special' memory needing heap_caps_malloc to allocate, or by fully integrating it making malloc() also able to return SPI RAM pointers.

Available options:

- Integrate RAM into memory map (SPIRAM_USE_MEMMAP)
- Make RAM allocatable using heap_caps_malloc(..., MALLOC_CAP_SPIRAM) (SPIRAM_USE_CAPS_ALLOC)
- Make RAM allocatable using malloc() as well (SPIRAM_USE_MALLOC)

CONFIG_SPIRAM_MEMTEST

Run memory test on SPI RAM initialization

Found in: [Component config](#) > [ESP32-specific](#) > [CONFIG_ESP32_SPIRAM_SUPPORT](#) > [SPI RAM config](#)

Runs a rudimentary memory test on initialization. Aborts when memory test fails. Disable this for slightly faster startup.

CONFIG_SPIRAM_MALLOC_ALWAYSINTERNAL

Maximum malloc() size, in bytes, to always put in internal memory

Found in: [Component config](#) > [ESP32-specific](#) > [CONFIG_ESP32_SPIRAM_SUPPORT](#) > [SPI RAM config](#)

If malloc() is capable of also allocating SPI-connected ram, its allocation strategy will prefer to allocate chunks less than this size in internal memory, while allocations larger than this will be done from external RAM. If allocation from the preferred region fails, an attempt is made to allocate from the non-preferred region instead, so malloc() will not suddenly fail when either internal or external memory is full.

CONFIG_SPIRAM_TRY_ALLOCATE_WIFI_LWIP

Try to allocate memories of WiFi and LWIP in SPIRAM firstly. If failed, allocate internal memory

Found in: [Component config](#) > [ESP32-specific](#) > [CONFIG_ESP32_SPIRAM_SUPPORT](#) > [SPI RAM config](#)

Try to allocate memories of WiFi and LWIP in SPIRAM firstly. If failed, try to allocate internal memory then.

CONFIG_SPIRAM_MALLOC_RESERVE_INTERNAL

Reserve this amount of bytes for data that specifically needs to be in DMA or internal memory

Found in: [Component config](#) > [ESP32-specific](#) > [CONFIG_ESP32_SPIRAM_SUPPORT](#) > [SPI RAM config](#)

Because the external/internal RAM allocation strategy is not always perfect, it sometimes may happen that the internal memory is entirely filled up. This causes allocations that are specifically done in internal memory, for example the stack for new tasks or memory to service DMA or have memory that's also available when SPI cache is down, to fail. This option reserves a pool specifically for requests like that; the memory in this pool is not given out when a normal malloc() is called.

Set this to 0 to disable this feature.

Note that because FreeRTOS stacks are forced to internal memory, they will also use this memory pool; be sure to keep this in mind when adjusting this value.

Note also that the DMA reserved pool may not be one single contiguous memory region, depending on the configured size and the static memory usage of the app.

CONFIG_SPIRAM_ALLOW_BSS_SEG_EXTERNAL_MEMORY

Allow .bss segment placed in external memory

Found in: [Component config](#) > [ESP32-specific](#) > [CONFIG_ESP32_SPIRAM_SUPPORT](#) > [SPI RAM config](#)

If enabled the option, and add EXT_RAM_ATTR defined your variable, then your variable will be placed in PSRAM instead of internal memory, and placed most of variables of lwip, net802.11, pp, blueandroid library to external memory defaultly.

CONFIG_SPIRAM_CACHE_WORKAROUND

Enable workaround for bug in SPI RAM cache for Rev1 ESP32s

Found in: [Component config](#) > [ESP32-specific](#) > [CONFIG_ESP32_SPIRAM_SUPPORT](#) > [SPI RAM config](#)

Revision 1 of the ESP32 has a bug that can cause a write to PSRAM not to take place in some situations when the cache line needs to be fetched from external RAM and an interrupt occurs. This enables a fix in the compiler (-mfix-esp32-psram-cache-issue) that makes sure the specific code that is vulnerable to this will not be emitted.

This will also not use any bits of newlib that are located in ROM, opting for a version that is compiled with the workaround and located in flash instead.

The workaround is not required for ESP32 revision 3 and above.

SPIRAM cache workaround debugging Contains:

- `CONFIG_SPIRAM_CACHE_WORKAROUND_STRATEGY`

CONFIG_SPIRAM_CACHE_WORKAROUND_STRATEGY

Workaround strategy

Found in: Component config > ESP32-specific > CONFIG_ESP32_SPIRAM_SUPPORT > SPI RAM config > SPIRAM cache workaround debugging

Select the workaround strategy. Note that the strategy for precompiled libraries (libgcc, newlib, bt, wifi) is not affected by this selection.

Unless you know you need a different strategy, it's suggested you stay with the default MEMW strategy. Note that DUPLDST can interfere with hardware encryption and this will be automatically disabled if this workaround is selected. 'Insert nops' is the workaround that was used in older esp-idf versions. This workaround still can cause faulty data transfers from/to SPI RAM in some situation.

Available options:

- Insert memw after vulnerable instructions (default) (SPIRAM_CACHE_WORKAROUND_STRATEGY_MEMW)
- Duplicate LD/ST for 32-bit, memw for 8/16 bit (SPIRAM_CACHE_WORKAROUND_STRATEGY_DUPLDST)
- Insert nops between vulnerable loads/stores (old strategy, obsolete) (SPIRAM_CACHE_WORKAROUND_STRATEGY_NOP)

CONFIG_SPIRAM_BANKSWITCH_ENABLE

Enable bank switching for >4MiB external RAM

Found in: Component config > ESP32-specific > CONFIG_ESP32_SPIRAM_SUPPORT > SPI RAM config

The ESP32 only supports 4MiB of external RAM in its address space. The hardware does support larger memories, but these have to be bank-switched in and out of this address space. Enabling this allows you to reserve some MMU pages for this, which allows the use of the esp_himem api to manage these banks.

#Note that this is limited to 62 banks, as esp_spiiram_writeback_cache needs some kind of mapping of #some banks below that mark to work. We cannot at this moment guarantee this to exist when himem is #enabled.

If spiram 2T mode is enabled, the size of 64Mbit psram will be changed as 32Mbit, so himem will be unusable.

CONFIG_SPIRAM_BANKSWITCH_RESERVE

Amount of 32K pages to reserve for bank switching

Found in: Component config > ESP32-specific > CONFIG_ESP32_SPIRAM_SUPPORT > SPI RAM config > CONFIG_SPIRAM_BANKSWITCH_ENABLE

Select the amount of banks reserved for bank switching. Note that the amount of RAM allocatable with malloc/esp_heap_alloc_caps will decrease by 32K for each page reserved here.

Note that this reservation is only actually done if your program actually uses the himem API. Without any himem calls, the reservation is not done and the original amount of memory will be available to malloc/esp_heap_alloc_caps.

CONFIG_SPIRAM_ALLOW_STACK_EXTERNAL_MEMORY

Allow external memory as an argument to xTaskCreateStatic

Found in: Component config > ESP32-specific > CONFIG_ESP32_SPIRAM_SUPPORT > SPI RAM config

Because some bits of the ESP32 code environment cannot be recompiled with the cache workaround, normally tasks cannot be safely run with their stack residing in external memory; for this reason `xTaskCreate` and friends always allocate stack in internal memory and `xTaskCreateStatic` will check if the memory passed to it is in internal memory. If you have a task that needs a large amount of stack and does not call on ROM code in any way (no direct calls, but also no Bluetooth/WiFi), you can try to disable this and use `xTaskCreateStatic` to create the tasks stack in external memory.

CONFIG_SPIRAM_OCCUPY_SPI_HOST

SPI host to use for 32MBit PSRAM

Found in: [Component config](#) > [ESP32-specific](#) > [CONFIG_ESP32_SPIRAM_SUPPORT](#) > [SPI RAM config](#)

When both flash and PSRAM is working under 80MHz, and the PSRAM is of type 32MBit, one of the HSPI/VSPI host will be used to output the clock. Select which one to use here.

Available options:

- HSPI host (SPI2) (`SPIRAM_OCCUPY_HSPI_HOST`)
- VSPI host (SPI3) (`SPIRAM_OCCUPY_VSPI_HOST`)
- Will not try to use any host, will abort if not able to use the PSRAM (`SPIRAM_OCCUPY_NO_HOST`)

PSRAM clock and cs IO for ESP32-D0WD Contains:

- [CONFIG_D0WD_PSRAM_CLK_IO](#)
- [CONFIG_D0WD_PSRAM_CS_IO](#)

CONFIG_D0WD_PSRAM_CLK_IO

PSRAM CLK IO number

Found in: [Component config](#) > [ESP32-specific](#) > [CONFIG_ESP32_SPIRAM_SUPPORT](#) > [SPI RAM config](#) > [PSRAM clock and cs IO for ESP32-D0WD](#)

The PSRAM CLOCK IO can be any unused GPIO, user can config it based on hardware design. If user use 1.8V flash and 1.8V psram, this value can only be one of 6, 7, 8, 9, 10, 11, 16, 17.

CONFIG_D0WD_PSRAM_CS_IO

PSRAM CS IO number

Found in: [Component config](#) > [ESP32-specific](#) > [CONFIG_ESP32_SPIRAM_SUPPORT](#) > [SPI RAM config](#) > [PSRAM clock and cs IO for ESP32-D0WD](#)

The PSRAM CS IO can be any unused GPIO, user can config it based on hardware design. If user use 1.8V flash and 1.8V psram, this value can only be one of 6, 7, 8, 9, 10, 11, 16, 17.

PSRAM clock and cs IO for ESP32-D2WD Contains:

- [CONFIG_D2WD_PSRAM_CLK_IO](#)
- [CONFIG_D2WD_PSRAM_CS_IO](#)

CONFIG_D2WD_PSRAM_CLK_IO

PSRAM CLK IO number

Found in: [Component config](#) > [ESP32-specific](#) > [CONFIG_ESP32_SPIRAM_SUPPORT](#) > [SPI RAM config](#) > [PSRAM clock and cs IO for ESP32-D2WD](#)

User can config it based on hardware design. For ESP32-D2WD chip, the psram can only be 1.8V psram, so this value can only be one of 6, 7, 8, 9, 10, 11, 16, 17.

CONFIG_D2WD_PSRAM_CS_IO

PSRAM CS IO number

Found in: Component config > ESP32-specific > CONFIG_ESP32_SPIRAM_SUPPORT > SPI RAM config > PSRAM clock and cs IO for ESP32-D2WD

User can config it based on hardware design. For ESP32-D2WD chip, the psram can only be 1.8V psram, so this value can only be one of 6, 7, 8, 9, 10, 11, 16, 17.

PSRAM clock and cs IO for ESP32-PICO Contains:

- *CONFIG_PICO_PSRAM_CS_IO*

CONFIG_PICO_PSRAM_CS_IO

PSRAM CS IO number

Found in: Component config > ESP32-specific > CONFIG_ESP32_SPIRAM_SUPPORT > SPI RAM config > PSRAM clock and cs IO for ESP32-PICO

The PSRAM CS IO can be any unused GPIO, user can config it based on hardware design.

For ESP32-PICO chip, the psram share clock with flash, so user do not need to configure the clock IO. For the reference hardware design, please refer to https://www.espressif.com/sites/default/files/documentation/esp32-pico-d4_datasheet_en.pdf

CONFIG_SPIRAM_CUSTOM_SPIWP_SD3_PIN

Use custom SPI PSRAM WP(SD3) Pin when flash pins set in eFuse (read help)

Found in: Component config > ESP32-specific > CONFIG_ESP32_SPIRAM_SUPPORT > SPI RAM config

This setting is only used if the SPI flash pins have been overridden by setting the eFuses SPI_PAD_CONFIG_XXX, and the SPI flash mode is DIO or DOUT.

When this is the case, the eFuse config only defines 3 of the 4 Quad I/O data pins. The WP pin (aka ESP32 pin “SD_DATA_3” or SPI flash pin “IO2”) is not specified in eFuse. The psram only has QPI mode, so a WP pin setting is necessary.

If this config item is set to N (default), the correct WP pin will be automatically used for any Espressif chip or module with integrated flash. If a custom setting is needed, set this config item to Y and specify the GPIO number connected to the WP pin.

When flash mode is set to QIO or QOUT, the PSRAM WP pin will be set the same as the SPI Flash WP pin configured in the bootloader.

CONFIG_SPIRAM_SPIWP_SD3_PIN

Custom SPI PSRAM WP(SD3) Pin

Found in: Component config > ESP32-specific > CONFIG_ESP32_SPIRAM_SUPPORT > SPI RAM config

The option “Use custom SPI PSRAM WP(SD3) pin” must be set or this value is ignored

If burning a customized set of SPI flash pins in eFuse and using DIO or DOUT mode for flash, set this value to the GPIO number of the SPIRAM WP pin.

CONFIG_SPIRAM_2T_MODE

Enable SPI PSRAM 2T mode

Found in: Component config > ESP32-specific > CONFIG_ESP32_SPIRAM_SUPPORT > SPI RAM config

Enable this option to fix single bit errors inside 64Mbit PSRAM.

Some 64Mbit PSRAM chips have a hardware issue in the RAM which causes bit errors at multiple fixed bit positions.

Note: If this option is enabled, the 64Mbit PSRAM chip will appear to be 32Mbit in size. Applications will not be affected unless the use the `esp_himem` APIs, which are not supported in 2T mode.

CONFIG_ESP32_TRAX

Use TRAX tracing feature

Found in: [Component config](#) > [ESP32-specific](#)

The ESP32 contains a feature which allows you to trace the execution path the processor has taken through the program. This is stored in a chunk of 32K (16K for single-processor) of memory that can't be used for general purposes anymore. Disable this if you do not know what this is.

CONFIG_ESP32_TRAX_TWOBANKS

Reserve memory for tracing both pro as well as app cpu execution

Found in: [Component config](#) > [ESP32-specific](#) > [CONFIG_ESP32_TRAX](#)

The ESP32 contains a feature which allows you to trace the execution path the processor has taken through the program. This is stored in a chunk of 32K (16K for single-processor) of memory that can't be used for general purposes anymore. Disable this if you do not know what this is.

Memory to reserve for trace, used in linker script

CONFIG_ESP32_UNIVERSAL_MAC_ADDRESSES

Number of universally administered (by IEEE) MAC address

Found in: [Component config](#) > [ESP32-specific](#)

Configure the number of universally administered (by IEEE) MAC addresses. During initialisation, MAC addresses for each network interface are generated or derived from a single base MAC address. If the number of universal MAC addresses is four, all four interfaces (WiFi station, WiFi softap, Bluetooth and Ethernet) receive a universally administered MAC address. These are generated sequentially by adding 0, 1, 2 and 3 (respectively) to the final octet of the base MAC address. If the number of universal MAC addresses is two, only two interfaces (WiFi station and Bluetooth) receive a universally administered MAC address. These are generated sequentially by adding 0 and 1 (respectively) to the base MAC address. The remaining two interfaces (WiFi softap and Ethernet) receive local MAC addresses. These are derived from the universal WiFi station and Bluetooth MAC addresses, respectively. When using the default (Espressif-assigned) base MAC address, either setting can be used. When using a custom universal MAC address range, the correct setting will depend on the allocation of MAC addresses in this range (either 2 or 4 per device.)

Available options:

- Two (`ESP32_UNIVERSAL_MAC_ADDRESSES_TWO`)
- Four (`ESP32_UNIVERSAL_MAC_ADDRESSES_FOUR`)

CONFIG_ESP32_ULP_COPROC_ENABLED

Enable Ultra Low Power (ULP) Coprocessor

Found in: [Component config](#) > [ESP32-specific](#)

Set to 'y' if you plan to load a firmware for the coprocessor.

If this option is enabled, further coprocessor configuration will appear in the Components menu.

CONFIG_ESP32_ULP_COPROC_RESERVE_MEM

RTC slow memory reserved for coprocessor

Found in: [Component config](#) > [ESP32-specific](#) > [CONFIG_ESP32_ULP_COPROC_ENABLED](#)

Bytes of memory to reserve for ULP coprocessor firmware & data.

Data is reserved at the beginning of RTC slow memory.

CONFIG_ESP32_PANIC

Panic handler behaviour

Found in: [Component config](#) > [ESP32-specific](#)

If FreeRTOS detects unexpected behaviour or an unhandled exception, the panic handler is invoked. Configure the panic handlers action here.

Available options:

- Print registers and halt (ESP32_PANIC_PRINT_HALT)
Outputs the relevant registers over the serial port and halt the processor. Needs a manual reset to restart.
- Print registers and reboot (ESP32_PANIC_PRINT_REBOOT)
Outputs the relevant registers over the serial port and immediately reset the processor.
- Silent reboot (ESP32_PANIC_SILENT_REBOOT)
Just resets the processor without outputting anything
- Invoke GDBStub (ESP32_PANIC_GDBSTUB)
Invoke gdbstub on the serial port, allowing for gdb to attach to it to do a postmortem of the crash.

CONFIG_ESP32_DEBUG_OCDAWARE

Make exception and panic handlers JTAG/OCD aware

Found in: [Component config](#) > [ESP32-specific](#)

The FreeRTOS panic and unhandled exception handlers can detect a JTAG OCD debugger and instead of panicking, have the debugger stop on the offending instruction.

CONFIG_ESP32_BROWNOUT_DET

Hardware brownout detect & reset

Found in: [Component config](#) > [ESP32-specific](#)

The ESP32 has a built-in brownout detector which can detect if the voltage is lower than a specific value. If this happens, it will reset the chip in order to prevent unintended behaviour.

CONFIG_ESP32_BROWNOUT_DET_LVL_SEL

Brownout voltage level

Found in: [Component config](#) > [ESP32-specific](#) > [CONFIG_ESP32_BROWNOUT_DET](#)

The brownout detector will reset the chip when the supply voltage is approximately below this level. Note that there may be some variation of brownout voltage level between each ESP32 chip.

#The voltage levels here are estimates, more work needs to be done to figure out the exact voltages #of the brownout threshold levels.

Available options:

- 2.43V +/- 0.05 (ESP32_BROWNOUT_DET_LVL_SEL_0)
- 2.48V +/- 0.05 (ESP32_BROWNOUT_DET_LVL_SEL_1)
- 2.58V +/- 0.05 (ESP32_BROWNOUT_DET_LVL_SEL_2)

- 2.62V +/- 0.05 (ESP32_BROWNOUT_DET_LVL_SEL_3)
- 2.67V +/- 0.05 (ESP32_BROWNOUT_DET_LVL_SEL_4)
- 2.70V +/- 0.05 (ESP32_BROWNOUT_DET_LVL_SEL_5)
- 2.77V +/- 0.05 (ESP32_BROWNOUT_DET_LVL_SEL_6)
- 2.80V +/- 0.05 (ESP32_BROWNOUT_DET_LVL_SEL_7)

CONFIG_ESP32_REDUCE_PHY_TX_POWER

Reduce PHY TX power when brownout reset

Found in: [Component config](#) > [ESP32-specific](#)

When brownout reset occurs, reduce PHY TX power to keep the code running

Note about the use of “FRC1” name: currently FRC1 timer is not used for # high resolution time-keeping anymore. Instead the esp_timer API, implemented # using FRC2 timer, is used. # FRC1 name in the option name is kept for compatibility.

CONFIG_ESP32_TIME_SYSCALL

Timers used for gettimeofday function

Found in: [Component config](#) > [ESP32-specific](#)

This setting defines which hardware timers are used to implement ‘gettimeofday’ and ‘time’ functions in C library.

- If both high-resolution and RTC timers are used, timekeeping will continue in deep sleep. Time will be reported at 1 microsecond resolution. This is the default, and the recommended option.
- If only high-resolution timer is used, gettimeofday will provide time at microsecond resolution. Time will not be preserved when going into deep sleep mode.
- If only RTC timer is used, timekeeping will continue in deep sleep, but time will be measured at 6.(6) microsecond resolution. Also the gettimeofday function itself may take longer to run.
- If no timers are used, gettimeofday and time functions return -1 and set errno to ENOSYS.
- When RTC is used for timekeeping, two RTC_STORE registers are used to keep time in deep sleep mode.

Available options:

- RTC and high-resolution timer (ESP32_TIME_SYSCALL_USE_RTC_FRC1)
- RTC (ESP32_TIME_SYSCALL_USE_RTC)
- High-resolution timer (ESP32_TIME_SYSCALL_USE_FRC1)
- None (ESP32_TIME_SYSCALL_USE_NONE)

CONFIG_ESP32_RTC_CLK_SRC

RTC clock source

Found in: [Component config](#) > [ESP32-specific](#)

Choose which clock is used as RTC clock source.

- “Internal 150kHz oscillator” option provides lowest deep sleep current consumption, and does not require extra external components. However frequency stability with respect to temperature is poor, so time may drift in deep/light sleep modes.
- “External 32kHz crystal” provides better frequency stability, at the expense of slightly higher (1uA) deep sleep current consumption.
- “External 32kHz oscillator” allows using 32kHz clock generated by an external circuit. In this case, external clock signal must be connected to 32K_XP pin. Amplitude should be <1.2V in case of sine wave signal, and <1V in case of square wave signal. Common mode voltage should be $0.1 < V_{cm} < 0.5V_{amp}$, where V_{amp} is the signal amplitude. Additionally, 1nF capacitor must be connected between 32K_XN pin and ground. 32K_XN pin can not be used as a GPIO in this case.

- “Internal 8.5MHz oscillator divided by 256” option results in higher deep sleep current (by 5uA) but has better frequency stability than the internal 150kHz oscillator. It does not require external components.

Available options:

- Internal 150kHz RC oscillator (ESP32_RTC_CLK_SRC_INT_RC)
- External 32kHz crystal (ESP32_RTC_CLK_SRC_EXT_CRYST)
- External 32kHz oscillator at 32K_XP pin (ESP32_RTC_CLK_SRC_EXT_OSC)
- Internal 8.5MHz oscillator, divided by 256 (~33kHz) (ESP32_RTC_CLK_SRC_INT_8MD256)

CONFIG_ESP32_RTC_EXT_CRYST_ADDIT_CURRENT

Additional current for external 32kHz crystal

Found in: [Component config](#) > [ESP32-specific](#)

Choose which additional current is used for rtc external crystal.

- With some 32kHz crystal configurations, the X32N and X32P pins may not have enough drive strength to keep the crystal oscillating during deep sleep. If this option is enabled, additional current from touchpad 9 is provided internally to drive the 32kHz crystal. If this option is enabled, deep sleep current is slightly higher (4-5uA) and the touchpad and ULP wakeup sources are not available.

CONFIG_ESP32_RTC_CLK_CAL_CYCLES

Number of cycles for RTC_SLOW_CLK calibration

Found in: [Component config](#) > [ESP32-specific](#)

When the startup code initializes RTC_SLOW_CLK, it can perform calibration by comparing the RTC_SLOW_CLK frequency with main XTAL frequency. This option sets the number of RTC_SLOW_CLK cycles measured by the calibration routine. Higher numbers increase calibration precision, which may be important for applications which spend a lot of time in deep sleep. Lower numbers reduce startup time.

When this option is set to 0, clock calibration will not be performed at startup, and approximate clock frequencies will be assumed:

- 150000 Hz if internal RC oscillator is used as clock source. For this use value 1024.
- 32768 Hz if the 32k crystal oscillator is used. For this use value 3000 or more. In case more value will help improve the definition of the launch of the crystal. If the crystal could not start, it will be switched to internal RC.

CONFIG_ESP32_RTC_XTAL_CAL_RETRY

Number of attempts to repeat 32k XTAL calibration

Found in: [Component config](#) > [ESP32-specific](#)

Number of attempts to repeat 32k XTAL calibration before giving up and switching to the internal RC. Increase this option if the 32k crystal oscillator does not start and switches to internal RC.

CONFIG_ESP32_RTC_XTAL_BOOTSTRAP_CYCLES

Bootstrap cycles for external 32kHz crystal

Found in: [Component config](#) > [ESP32-specific](#)

To reduce the startup time of an external RTC crystal, we bootstrap it with a 32kHz square wave for a fixed number of cycles. Setting 0 will disable bootstrapping (if disabled, the crystal may take longer to start up or fail to oscillate under some conditions).

If this value is too high, a faulty crystal may initially start and then fail. If this value is too low, an otherwise good crystal may not start.

To accurately determine if the crystal has started, set a larger “Number of cycles for RTC_SLOW_CLK calibration” (about 3000).

CONFIG_ESP32_DEEP_SLEEP_WAKEUP_DELAY

Extra delay in deep sleep wake stub (in us)

Found in: [Component config](#) > [ESP32-specific](#)

When ESP32 exits deep sleep, the CPU and the flash chip are powered on at the same time. CPU will run deep sleep stub first, and then proceed to load code from flash. Some flash chips need sufficient time to pass between power on and first read operation. By default, without any extra delay, this time is approximately 900us, although some flash chip types need more than that.

By default extra delay is set to 2000us. When optimizing startup time for applications which require it, this value may be reduced.

If you are seeing “flash read err, 1000” message printed to the console after deep sleep reset, try increasing this value.

CONFIG_ESP32_XTAL_FREQ_SEL

Main XTAL frequency

Found in: [Component config](#) > [ESP32-specific](#)

ESP32 currently supports the following XTAL frequencies:

- 26 MHz
- 40 MHz

Startup code can automatically estimate XTAL frequency. This feature uses the internal 8MHz oscillator as a reference. Because the internal oscillator frequency is temperature dependent, it is not recommended to use automatic XTAL frequency detection in applications which need to work at high ambient temperatures and use high-temperature qualified chips and modules.

Available options:

- 40 MHz (ESP32_XTAL_FREQ_40)
- 26 MHz (ESP32_XTAL_FREQ_26)
- Autodetect (ESP32_XTAL_FREQ_AUTO)

CONFIG_ESP32_DISABLE_BASIC_ROM_CONSOLE

Permanently disable BASIC ROM Console

Found in: [Component config](#) > [ESP32-specific](#)

If set, the first time the app boots it will disable the BASIC ROM Console permanently (by burning an eFuse).

Otherwise, the BASIC ROM Console starts on reset if no valid bootloader is read from the flash.

(Enabling secure boot also disables the BASIC ROM Console by default.)

CONFIG_ESP32_NO_BLOBS

No Binary Blobs

Found in: [Component config](#) > [ESP32-specific](#)

If enabled, this disables the linking of binary libraries in the application build. Note that after enabling this Wi-Fi/Bluetooth will not work.

CONFIG_ESP32_COMPATIBLE_PRE_V2_1_BOOTLOADERS

App compatible with bootloaders before ESP-IDF v2.1

Found in: [Component config](#) > [ESP32-specific](#)

Bootloaders before ESP-IDF v2.1 did less initialisation of the system clock. This setting needs to be enabled to build an app which can be booted by these older bootloaders.

If this setting is enabled, the app can be booted by any bootloader from IDF v1.0 up to the current version.

If this setting is disabled, the app can only be booted by bootloaders from IDF v2.1 or newer.

Enabling this setting adds approximately 1KB to the app's IRAM usage.

CONFIG_ESP32_COMPATIBLE_PRE_V3_1_BOOTLOADERS

App compatible with bootloader and partition table before ESP-IDF v3.1

Found in: [Component config](#) > [ESP32-specific](#)

Partition tables before ESP-IDF V3.1 do not contain an MD5 checksum field, and the bootloader before ESP-IDF v3.1 cannot read a partition table that contains an MD5 checksum field.

Enable this option only if your app needs to boot on a bootloader and/or partition table that was generated from a version *before* ESP-IDF v3.1.

If this option and Flash Encryption are enabled at the same time, and any data partitions in the partition table are marked Encrypted, then the partition encrypted flag should be manually verified in the app before accessing the partition (see CVE-2021-27926).

CONFIG_ESP32_RTCDATA_IN_FAST_MEM

Place RTC_DATA_ATTR and RTC_RODATA_ATTR variables into RTC fast memory segment

Found in: [Component config](#) > [ESP32-specific](#)

This option allows to place .rtc_data and .rtc_rodata sections into RTC fast memory segment to free the slow memory region for ULP programs. This option depends on the CONFIG_FREERTOS_UNICORE option because RTC fast memory can be accessed only by PRO_CPU core.

CONFIG_ESP32_USE_FIXED_STATIC_RAM_SIZE

Use fixed static RAM size

Found in: [Component config](#) > [ESP32-specific](#)

If this option is disabled, the DRAM part of the heap starts right after the .bss section, within the dram0_0 region. As a result, adding or removing some static variables will change the available heap size.

If this option is enabled, the DRAM part of the heap starts right after the dram0_0 region, where its length is set with ESP32_FIXED_STATIC_RAM_SIZE

CONFIG_ESP32_FIXED_STATIC_RAM_SIZE

Fixed Static RAM size

Found in: [Component config](#) > [ESP32-specific](#) > [CONFIG_ESP32_USE_FIXED_STATIC_RAM_SIZE](#)

RAM size dedicated for static variables (.data & .bss sections). Please note that the actual length will be reduced by BT_RESERVE_DRAM if Bluetooth controller is enabled.

CONFIG_ESP32_DPORT_DIS_INTERRUPT_LVL

Disable the interrupt level for the DPORT workarounds

Found in: [Component config](#) > [ESP32-specific](#)

To prevent interrupting DPORT workarounds, need to disable interrupt with a maximum used level in the system.

Power Management Contains:

- [CONFIG_PM_ENABLE](#)

CONFIG_PM_ENABLE

Support for power management

Found in: [Component config](#) > [Power Management](#)

If enabled, application is compiled with support for power management. This option has run-time overhead (increased interrupt latency, longer time to enter idle state), and it also reduces accuracy of RTOS ticks and timers used for timekeeping. Enable this option if application uses power management APIs.

CONFIG_PM_DFS_INIT_AUTO

Enable dynamic frequency scaling (DFS) at startup

Found in: [Component config](#) > [Power Management](#) > [CONFIG_PM_ENABLE](#)

If enabled, startup code configures dynamic frequency scaling. Max CPU frequency is set to CONFIG_ESP32_DEFAULT_CPU_FREQ_MHZ setting, min frequency is set to XTAL frequency. If disabled, DFS will not be active until the application configures it using esp_pm_configure function.

CONFIG_PM_USE_RTC_TIMER_REF

Use RTC timer to prevent time drift (EXPERIMENTAL)

Found in: [Component config](#) > [Power Management](#) > [CONFIG_PM_ENABLE](#)

When APB clock frequency changes, high-resolution timer (esp_timer) scale and base value need to be adjusted. Each adjustment may cause small error, and over time such small errors may cause time drift. If this option is enabled, RTC timer will be used as a reference to compensate for the drift. It is recommended that this option is only used if 32k XTAL is selected as RTC clock source.

CONFIG_PM_PROFILING

Enable profiling counters for PM locks

Found in: [Component config](#) > [Power Management](#) > [CONFIG_PM_ENABLE](#)

If enabled, esp_pm_* functions will keep track of the amount of time each of the power management locks has been held, and esp_pm_dump_locks function will print this information. This feature can be used to analyze which locks are preventing the chip from going into a lower power state, and see what time the chip spends in each power saving mode. This feature does incur some run-time overhead, so should typically be disabled in production builds.

CONFIG_PM_TRACE

Enable debug tracing of PM using GPIOs

Found in: [Component config](#) > [Power Management](#) > [CONFIG_PM_ENABLE](#)

If enabled, some GPIOs will be used to signal events such as RTOS ticks, frequency switching, entry/exit from idle state. Refer to pm_trace.c file for the list of GPIOs. This feature is intended to be used when analyzing/debugging behavior of power management implementation, and should be kept disabled in applications.

ADC-Calibration Contains:

- [CONFIG_ADC_CAL_EFUSE_TP_ENABLE](#)
- [CONFIG_ADC_CAL_EFUSE_VREF_ENABLE](#)
- [CONFIG_ADC_CAL_LUT_ENABLE](#)

CONFIG_ADC_CAL_EFUSE_TP_ENABLE

Use Two Point Values

Found in: [Component config](#) > [ADC-Calibration](#)

Some ESP32s have Two Point calibration values burned into eFuse BLOCK3. This option will allow the ADC calibration component to characterize the ADC-Voltage curve using Two Point values if they are available.

CONFIG_ADC_CAL_EFUSE_VREF_ENABLE

Use eFuse Vref

Found in: [Component config](#) > [ADC-Calibration](#)

Some ESP32s have Vref burned into eFuse BLOCK0. This option will allow the ADC calibration component to characterize the ADC-Voltage curve using eFuse Vref if it is available.

CONFIG_ADC_CAL_LUT_ENABLE

Use Lookup Tables

Found in: [Component config](#) > [ADC-Calibration](#)

This option will allow the ADC calibration component to use Lookup Tables to correct for non-linear behavior in 11db attenuation. Other attenuations do not exhibit non-linear behavior hence will not be affected by this option.

Common ESP-related Contains:

- [CONFIG_ESP_TIMER_PROFILING](#)
- [CONFIG_ESP_ERR_TO_NAME_LOOKUP](#)
- [CONFIG_ESP_SYSTEM_EVENT_QUEUE_SIZE](#)
- [CONFIG_ESP_SYSTEM_EVENT_TASK_STACK_SIZE](#)
- [CONFIG_ESP_MAIN_TASK_STACK_SIZE](#)
- [CONFIG_ESP_IPC_TASK_STACK_SIZE](#)
- [CONFIG_ESP_IPC_USES_CALLERS_PRIORITY](#)
- [CONFIG_ESP_TIMER_TASK_STACK_SIZE](#)
- [CONFIG_ESP_MINIMAL_SHARED_STACK_SIZE](#)
- [CONFIG_ESP_CONSOLE_UART](#)
- [CONFIG_ESP_CONSOLE_UART_NUM](#)
- [CONFIG_ESP_CONSOLE_UART_TX_GPIO](#)
- [CONFIG_ESP_CONSOLE_UART_RX_GPIO](#)
- [CONFIG_ESP_CONSOLE_UART_BAUDRATE](#)

- [CONFIG_ESP_INT_WDT](#)
- [CONFIG_ESP_TASK_WDT](#)
- [CONFIG_ESP_PANIC_HANDLER_IRAM](#)

CONFIG_ESP_TIMER_PROFILING

Enable esp_timer profiling features

Found in: [Component config](#) > [Common ESP-related](#)

If enabled, esp_timer_dump will dump information such as number of times the timer was started, number of times the timer has triggered, and the total time it took for the callback to run. This option has some effect on timer performance and the amount of memory used for timer storage, and should only be used for debugging/testing purposes.

CONFIG_ESP_ERR_TO_NAME_LOOKUP

Enable lookup of error code strings

Found in: [Component config](#) > [Common ESP-related](#)

Functions esp_err_to_name() and esp_err_to_name_r() return string representations of error codes from a pre-generated lookup table. This option can be used to turn off the use of the look-up table in order to save memory but this comes at the price of sacrificing distinguishable (meaningful) output string representations.

CONFIG_ESP_SYSTEM_EVENT_QUEUE_SIZE

System event queue size

Found in: [Component config](#) > [Common ESP-related](#)

Config system event queue size in different application.

CONFIG_ESP_SYSTEM_EVENT_TASK_STACK_SIZE

Event loop task stack size

Found in: [Component config](#) > [Common ESP-related](#)

Config system event task stack size in different application.

CONFIG_ESP_MAIN_TASK_STACK_SIZE

Main task stack size

Found in: [Component config](#) > [Common ESP-related](#)

Configure the “main task” stack size. This is the stack of the task which calls app_main(). If app_main() returns then this task is deleted and its stack memory is freed.

CONFIG_ESP_IPC_TASK_STACK_SIZE

Inter-Processor Call (IPC) task stack size

Found in: [Component config](#) > [Common ESP-related](#)

Configure the IPC tasks stack size. One IPC task runs on each core (in dual core mode), and allows for cross-core function calls.

See IPC documentation for more details.

The default stack size should be enough for most common use cases. It can be shrunk if you are sure that you do not use any custom IPC functionality.

CONFIG_ESP_IPC_USES_CALLERS_PRIORITY

IPC runs at caller's priority

Found in: [Component config](#) > [Common ESP-related](#)

If this option is not enabled then the IPC task will keep behavior same as prior to that of ESP-IDF v4.0, and hence IPC task will run at (configMAX_PRIORITIES - 1) priority.

CONFIG_ESP_TIMER_TASK_STACK_SIZE

High-resolution timer task stack size

Found in: [Component config](#) > [Common ESP-related](#)

Configure the stack size of esp_timer/ets_timer task. This task is used to dispatch callbacks of timers created using ets_timer and esp_timer APIs. If you are seeing stack overflow errors in timer task, increase this value.

Note that this is not the same as FreeRTOS timer task. To configure FreeRTOS timer task size, see "FreeRTOS timer task stack size" option in "FreeRTOS" menu.

CONFIG_ESP_MINIMAL_SHARED_STACK_SIZE

Minimal allowed size for shared stack

Found in: [Component config](#) > [Common ESP-related](#)

Minimal value of size, in bytes, accepted to execute an expression with shared stack.

CONFIG_ESP_CONSOLE_UART

UART for console output

Found in: [Component config](#) > [Common ESP-related](#)

Select whether to use UART for console output (through stdout and stderr).

- Default is to use UART0 on pins GPIO1(TX) and GPIO3(RX).
- If "Custom" is selected, UART0 or UART1 can be chosen, and any pins can be selected.
- If "None" is selected, there will be no console output on any UART, except for initial output from ROM bootloader. This output can be further suppressed by bootstrapping GPIO13 pin to low logic level.

Available options:

- Default: UART0, TX=GPIO1, RX=GPIO3 (ESP_CONSOLE_UART_DEFAULT)
- Custom (ESP_CONSOLE_UART_CUSTOM)
- None (ESP_CONSOLE_UART_NONE)

CONFIG_ESP_CONSOLE_UART_NUM

UART peripheral to use for console output (0-1)

Found in: [Component config](#) > [Common ESP-related](#)

Due to a ROM bug, UART2 is not supported for console output via ets_printf.

Available options:

- UART0 (ESP_CONSOLE_UART_CUSTOM_NUM_0)
- UART1 (ESP_CONSOLE_UART_CUSTOM_NUM_1)

CONFIG_ESP_CONSOLE_UART_TX_GPIO

UART TX on GPIO#

Found in: [Component config](#) > [Common ESP-related](#)

CONFIG_ESP_CONSOLE_UART_RX_GPIO

UART RX on GPIO#

Found in: [Component config](#) > [Common ESP-related](#)

CONFIG_ESP_CONSOLE_UART_BAUDRATE

UART console baud rate

Found in: [Component config](#) > [Common ESP-related](#)

CONFIG_ESP_INT_WDT

Interrupt watchdog

Found in: [Component config](#) > [Common ESP-related](#)

This watchdog timer can detect if the FreeRTOS tick interrupt has not been called for a certain time, either because a task turned off interrupts and did not turn them on for a long time, or because an interrupt handler did not return. It will try to invoke the panic handler first and failing that reset the SoC.

CONFIG_ESP_INT_WDT_TIMEOUT_MS

Interrupt watchdog timeout (ms)

Found in: [Component config](#) > [Common ESP-related](#) > [CONFIG_ESP_INT_WDT](#)

The timeout of the watchdog, in milliseconds. Make this higher than the FreeRTOS tick rate.

CONFIG_ESP_INT_WDT_CHECK_CPU1

Also watch CPU1 tick interrupt

Found in: [Component config](#) > [Common ESP-related](#) > [CONFIG_ESP_INT_WDT](#)

Also detect if interrupts on CPU 1 are disabled for too long.

CONFIG_ESP_TASK_WDT

Initialize Task Watchdog Timer on startup

Found in: [Component config](#) > [Common ESP-related](#)

The Task Watchdog Timer can be used to make sure individual tasks are still running. Enabling this option will cause the Task Watchdog Timer to be initialized automatically at startup. The Task Watchdog timer can be initialized after startup as well (see Task Watchdog Timer API Reference)

CONFIG_ESP_TASK_WDT_PANIC

Invoke panic handler on Task Watchdog timeout

Found in: [Component config](#) > [Common ESP-related](#) > [CONFIG_ESP_TASK_WDT](#)

If this option is enabled, the Task Watchdog Timer will be configured to trigger the panic handler when it times out. This can also be configured at run time (see Task Watchdog Timer API Reference)

CONFIG_ESP_TASK_WDT_TIMEOUT_S

Task Watchdog timeout period (seconds)

Found in: [Component config](#) > [Common ESP-related](#) > [CONFIG_ESP_TASK_WDT](#)

Timeout period configuration for the Task Watchdog Timer in seconds. This is also configurable at run time (see Task Watchdog Timer API Reference)

CONFIG_ESP_TASK_WDT_CHECK_IDLE_TASK_CPU0

Watch CPU0 Idle Task

Found in: [Component config](#) > [Common ESP-related](#) > [CONFIG_ESP_TASK_WDT](#)

If this option is enabled, the Task Watchdog Timer will watch the CPU0 Idle Task. Having the Task Watchdog watch the Idle Task allows for detection of CPU starvation as the Idle Task not being called is usually a symptom of CPU starvation. Starvation of the Idle Task is detrimental as FreeRTOS household tasks depend on the Idle Task getting some runtime every now and then.

CONFIG_ESP_TASK_WDT_CHECK_IDLE_TASK_CPU1

Watch CPU1 Idle Task

Found in: [Component config](#) > [Common ESP-related](#) > [CONFIG_ESP_TASK_WDT](#)

If this option is enabled, the Task Watchdog Timer will watch the CPU1 Idle Task.

CONFIG_ESP_PANIC_HANDLER_IRAM

Place panic handler code in IRAM

Found in: [Component config](#) > [Common ESP-related](#)

If this option is disabled (default), the panic handler code is placed in flash not IRAM. This means that if ESP-IDF crashes while flash cache is disabled, the panic handler will automatically re-enable flash cache before running GDB Stub or Core Dump. This adds some minor risk, if the flash cache status is also corrupted during the crash.

If this option is enabled, the panic handler code is placed in IRAM. This allows the panic handler to run without needing to re-enable cache first. This may be necessary to debug some complex issues with crashes while flash cache is disabled (for example, when writing to SPI flash.)

Ethernet Contains:

- [CONFIG_ETH_USE_ESP32_EMAC](#)
- [CONFIG_ETH_USE_SPI_ETHERNET](#)
- [CONFIG_ETH_USE_OPENETH](#)

CONFIG_ETH_USE_ESP32_EMAC

Support ESP32 internal EMAC controller

Found in: [Component config](#) > [Ethernet](#)

ESP32 integrates a 10/100M Ethernet MAC controller.

Contains:

- [CONFIG_ETH_PHY_INTERFACE](#)
- [CONFIG_ETH_RMII_CLK_MODE](#)
- [CONFIG_ETH_RMII_CLK_OUTPUT_GPIO0](#)
- [CONFIG_ETH_RMII_CLK_OUT_GPIO](#)
- [CONFIG_ETH_DMA_BUFFER_SIZE](#)

- [CONFIG_ETH_DMA_RX_BUFFER_NUM](#)
- [CONFIG_ETH_DMA_TX_BUFFER_NUM](#)

CONFIG_ETH_PHY_INTERFACE

PHY interface

Found in: [Component config](#) > [Ethernet](#) > [CONFIG_ETH_USE_ESP32_EMAC](#)

Select the communication interface between MAC and PHY chip.

Available options:

- Reduced Media Independent Interface (RMII) ([ETH_PHY_INTERFACE_RMII](#))
- Media Independent Interface (MII) ([ETH_PHY_INTERFACE_MII](#))

CONFIG_ETH_RMII_CLK_MODE

RMII clock mode

Found in: [Component config](#) > [Ethernet](#) > [CONFIG_ETH_USE_ESP32_EMAC](#)

Select external or internal RMII clock.

Available options:

- Input RMII clock from external ([ETH_RMII_CLK_INPUT](#))
MAC will get RMII clock from outside. Note that ESP32 only supports GPIO0 to input the RMII clock.
- Output RMII clock from internal ([ETH_RMII_CLK_OUTPUT](#))
ESP32 can generate RMII clock by internal APLL. This clock can be routed to the external PHY device. ESP32 supports to route the RMII clock to GPIO0/16/17.

CONFIG_ETH_RMII_CLK_OUTPUT_GPIO0

Output RMII clock from GPIO0 (Experimental!)

Found in: [Component config](#) > [Ethernet](#) > [CONFIG_ETH_USE_ESP32_EMAC](#)

GPIO0 can be set to output a pre-divided PLL clock (test only!). Enabling this option will configure GPIO0 to output a 50MHz clock. In fact this clock doesn't have directly relationship with EMAC peripheral. Sometimes this clock won't work well with your PHY chip. You might need to add some extra devices after GPIO0 (e.g. inverter). Note that outputting RMII clock on GPIO0 is an experimental practice. If you want the Ethernet to work with WiFi, don't select GPIO0 output mode for stability.

CONFIG_ETH_RMII_CLK_OUT_GPIO

RMII clock GPIO number

Found in: [Component config](#) > [Ethernet](#) > [CONFIG_ETH_USE_ESP32_EMAC](#)

Set the GPIO number to output RMII Clock.

CONFIG_ETH_DMA_BUFFER_SIZE

Ethernet DMA buffer size (Byte)

Found in: [Component config](#) > [Ethernet](#) > [CONFIG_ETH_USE_ESP32_EMAC](#)

Set the size of each buffer used by Ethernet MAC DMA.

CONFIG_ETH_DMA_RX_BUFFER_NUM

Amount of Ethernet DMA Rx buffers

Found in: [Component config](#) > [Ethernet](#) > [CONFIG_ETH_USE_ESP32_EMAC](#)

Number of DMA receive buffers. Each buffer's size is ETH_DMA_BUFFER_SIZE. Larger number of buffers could increase throughput somehow.

CONFIG_ETH_DMA_TX_BUFFER_NUM

Amount of Ethernet DMA Tx buffers

Found in: [Component config](#) > [Ethernet](#) > [CONFIG_ETH_USE_ESP32_EMAC](#)

Number of DMA transmit buffers. Each buffer's size is ETH_DMA_BUFFER_SIZE. Larger number of buffers could increase throughput somehow.

CONFIG_ETH_USE_SPI_ETHERNET

Support SPI to Ethernet Module

Found in: [Component config](#) > [Ethernet](#)

ESP-IDF can also support some SPI-Ethernet modules.

Contains:

- [CONFIG_ETH_SPI_ETHERNET_DM9051](#)

CONFIG_ETH_SPI_ETHERNET_DM9051

Use DM9051

Found in: [Component config](#) > [Ethernet](#) > [CONFIG_ETH_USE_SPI_ETHERNET](#)

DM9051 is a fast Ethernet controller with an SPI interface. It's also integrated with a 10/100M PHY and MAC. Set true to enable DM9051 driver.

CONFIG_ETH_USE_OPENETH

Support OpenCores Ethernet MAC (for use with QEMU)

Found in: [Component config](#) > [Ethernet](#)

OpenCores Ethernet MAC driver can be used when an ESP-IDF application is executed in QEMU. This driver is not supported when running on a real chip.

Contains:

- [CONFIG_ETH_OPENETH_DMA_RX_BUFFER_NUM](#)
- [CONFIG_ETH_OPENETH_DMA_TX_BUFFER_NUM](#)

CONFIG_ETH_OPENETH_DMA_RX_BUFFER_NUM

Number of Ethernet DMA Rx buffers

Found in: [Component config](#) > [Ethernet](#) > [CONFIG_ETH_USE_OPENETH](#)

Number of DMA receive buffers, each buffer is 1600 bytes.

CONFIG_ETH_OPENETH_DMA_TX_BUFFER_NUM

Number of Ethernet DMA Tx buffers

Found in: [Component config](#) > [Ethernet](#) > [CONFIG_ETH_USE_OPENETH](#)

Number of DMA transmit buffers, each buffer is 1600 bytes.

Event Loop Library Contains:

- [CONFIG_ESP_EVENT_LOOP_PROFILING](#)
- [CONFIG_ESP_EVENT_POST_FROM_ISR](#)

CONFIG_ESP_EVENT_LOOP_PROFILING

Enable event loop profiling

Found in: [Component config](#) > [Event Loop Library](#)

Enables collections of statistics in the event loop library such as the number of events posted to/received by an event loop, number of callbacks involved, number of events dropped to a full event loop queue, run time of event handlers, and number of times/run time of each event handler.

CONFIG_ESP_EVENT_POST_FROM_ISR

Support posting events from ISRs

Found in: [Component config](#) > [Event Loop Library](#)

Enable posting events from interrupt handlers.

CONFIG_ESP_EVENT_POST_FROM_IRAM_ISR

Support posting events from ISRs placed in IRAM

Found in: [Component config](#) > [Event Loop Library](#) > [CONFIG_ESP_EVENT_POST_FROM_ISR](#)

Enable posting events from interrupt handlers placed in IRAM. Enabling this option places API functions `esp_event_post` and `esp_event_post_to` in IRAM.

GDB Stub Contains:

- [CONFIG_ESP_GDBSTUB_SUPPORT_TASKS](#)

CONFIG_ESP_GDBSTUB_SUPPORT_TASKS

Enable listing FreeRTOS tasks through GDB Stub

Found in: [Component config](#) > [GDB Stub](#)

If enabled, GDBStub can supply the list of FreeRTOS tasks to GDB. Thread list can be queried from GDB using 'info threads' command. Note that if GDB task lists were corrupted, this feature may not work. If GDBStub fails, try disabling this feature.

CONFIG_ESP_GDBSTUB_MAX_TASKS

Maximum number of tasks supported by GDB Stub

Found in: [Component config](#) > [GDB Stub](#) > [CONFIG_ESP_GDBSTUB_SUPPORT_TASKS](#)

Set the number of tasks which GDB Stub will support.

ESP HTTP client Contains:

- [CONFIG_ESP_HTTP_CLIENT_ENABLE_HTTPS](#)
- [CONFIG_ESP_HTTP_CLIENT_ENABLE_BASIC_AUTH](#)

CONFIG_ESP_HTTP_CLIENT_ENABLE_HTTPS

Enable https

Found in: [Component config](#) > [ESP HTTP client](#)

This option will enable https protocol by linking mbedtls library and initializing SSL transport

CONFIG_ESP_HTTP_CLIENT_ENABLE_BASIC_AUTH

Enable HTTP Basic Authentication

Found in: [Component config](#) > [ESP HTTP client](#)

This option will enable HTTP Basic Authentication. It is disabled by default as Basic auth uses unencrypted encoding, so it introduces a vulnerability when not using TLS

HTTP Server Contains:

- [CONFIG_HTTPD_MAX_REQ_HDR_LEN](#)
- [CONFIG_HTTPD_MAX_URI_LEN](#)
- [CONFIG_HTTPD_ERR_RESP_NO_DELAY](#)
- [CONFIG_HTTPD_PURGE_BUF_LEN](#)
- [CONFIG_HTTPD_LOG_PURGE_DATA](#)

CONFIG_HTTPD_MAX_REQ_HDR_LEN

Max HTTP Request Header Length

Found in: [Component config](#) > [HTTP Server](#)

This sets the maximum supported size of headers section in HTTP request packet to be processed by the server

CONFIG_HTTPD_MAX_URI_LEN

Max HTTP URI Length

Found in: [Component config](#) > [HTTP Server](#)

This sets the maximum supported size of HTTP request URI to be processed by the server

CONFIG_HTTPD_ERR_RESP_NO_DELAY

Use TCP_NODELAY socket option when sending HTTP error responses

Found in: [Component config](#) > [HTTP Server](#)

Using TCP_NODELAY socket option ensures that HTTP error response reaches the client before the underlying socket is closed. Please note that turning this off may cause multiple test failures

CONFIG_HTTPD_PURGE_BUF_LEN

Length of temporary buffer for purging data

Found in: [Component config](#) > [HTTP Server](#)

This sets the size of the temporary buffer used to receive and discard any remaining data that is received from the HTTP client in the request, but not processed as part of the server HTTP request handler.

If the remaining data is larger than the available buffer size, the buffer will be filled in multiple iterations. The buffer should be small enough to fit on the stack, but large enough to avoid excessive iterations.

CONFIG_HTTPD_LOG_PURGE_DATA

Log purged content data at Debug level

Found in: [Component config](#) > [HTTP Server](#)

Enabling this will log discarded binary HTTP request data at Debug level. For large content data this may not be desirable as it will clutter the log.

ESP HTTPS OTA Contains:

- [CONFIG_OTA_ALLOW_HTTP](#)

CONFIG_OTA_ALLOW_HTTP

Allow HTTP for OTA (WARNING: ONLY FOR TESTING PURPOSE, READ HELP)

Found in: [Component config](#) > [ESP HTTPS OTA](#)

It is highly recommended to keep HTTPS (along with server certificate validation) enabled. Enabling this option comes with potential risk of: - Non-encrypted communication channel with server - Accepting firmware upgrade image from server with fake identity

ESP HTTPS server Contains:

- [CONFIG_ESP_HTTPS_SERVER_ENABLE](#)

CONFIG_ESP_HTTPS_SERVER_ENABLE

Enable ESP_HTTPS_SERVER component

Found in: [Component config](#) > [ESP HTTPS server](#)

Enable ESP HTTPS server component

ESP NETIF Adapter Contains:

- [CONFIG_ESP_NETIF_IP_LOST_TIMER_INTERVAL](#)
- [CONFIG_ESP_NETIF_USE_TCPIP_STACK_LIB](#)
- [CONFIG_ESP_NETIF_TCPIP_ADAPTER_COMPATIBLE_LAYER](#)

CONFIG_ESP_NETIF_IP_LOST_TIMER_INTERVAL

IP Address lost timer interval (seconds)

Found in: [Component config](#) > [ESP NETIF Adapter](#)

The value of 0 indicates the IP lost timer is disabled, otherwise the timer is enabled.

The IP address may be lost because of some reasons, e.g. when the station disconnects from soft-AP, or when DHCP IP renew fails etc. If the IP lost timer is enabled, it will be started everytime the IP is

lost. Event SYSTEM_EVENT_STA_LOST_IP will be raised if the timer expires. The IP lost timer is stopped if the station get the IP again before the timer expires.

CONFIG_ESP_NETIF_USE_TCPIP_STACK_LIB

TCP/IP Stack Library

Found in: Component config > ESP NETIF Adapter

Choose the TCP/IP Stack to work, for example, LwIP, uIP, etc.

Available options:

- LwIP (ESP_NETIF_TCPIP_LWIP)
lwIP is a small independent implementation of the TCP/IP protocol suite.
- Loopback (ESP_NETIF_LOOPBACK)
Dummy implementation of esp-netif functionality which connects driver transmit to receive function. This option is for testing purpose only

CONFIG_ESP_NETIF_TCPIP_ADAPTER_COMPATIBLE_LAYER

Enable backward compatible tcpip_adapter interface

Found in: Component config > ESP NETIF Adapter

Backward compatible interface to tcpip_adapter is enabled by default to support legacy TCP/IP stack initialisation code. Disable this option to use only esp-netif interface.

Wi-Fi Contains:

- *CONFIG_ESP32_WIFI_SW_COEXIST_ENABLE*
- *CONFIG_ESP32_WIFI_STATIC_RX_BUFFER_NUM*
- *CONFIG_ESP32_WIFI_DYNAMIC_RX_BUFFER_NUM*
- *CONFIG_ESP32_WIFI_TX_BUFFER*
- *CONFIG_ESP32_WIFI_STATIC_TX_BUFFER_NUM*
- *CONFIG_ESP32_WIFI_DYNAMIC_TX_BUFFER_NUM*
- *CONFIG_ESP32_WIFI_CSI_ENABLED*
- *CONFIG_ESP32_WIFI_AMPDU_TX_ENABLED*
- *CONFIG_ESP32_WIFI_AMPDU_RX_ENABLED*
- *CONFIG_ESP32_WIFI_NVS_ENABLED*
- *CONFIG_ESP32_WIFI_TASK_CORE_ID*
- *CONFIG_ESP32_WIFI_SOFTAP_BEACON_MAX_LEN*
- *CONFIG_ESP32_WIFI_MGMT_SBUF_NUM*
- *CONFIG_ESP32_WIFI_DEBUG_LOG_ENABLE*
- *CONFIG_ESP32_WIFI_IRAM_OPT*
- *CONFIG_ESP32_WIFI_RX_IRAM_OPT*
- *CONFIG_ESP32_WIFI_ENABLE_WPA3_SAE*

CONFIG_ESP32_WIFI_SW_COEXIST_ENABLE

Software controls WiFi/Bluetooth coexistence

Found in: Component config > Wi-Fi

If enabled, WiFi & Bluetooth coexistence is controlled by software rather than hardware. Recommended for heavy traffic scenarios. Both coexistence configuration options are automatically managed, no user intervention is required. If only Bluetooth is used, it is recommended to disable this option to reduce binary file size.

CONFIG_ESP32_WIFI_STATIC_RX_BUFFER_NUM

Max number of WiFi static RX buffers

Found in: [Component config](#) > [Wi-Fi](#)

Set the number of WiFi static RX buffers. Each buffer takes approximately 1.6KB of RAM. The static rx buffers are allocated when `esp_wifi_init` is called, they are not freed until `esp_wifi_deinit` is called.

WiFi hardware use these buffers to receive all 802.11 frames. A higher number may allow higher throughput but increases memory use. If `ESP32_WIFI_AMPDU_RX_ENABLED` is enabled, this value is recommended to set equal or bigger than `ESP32_WIFI_RX_BA_WIN` in order to achieve better throughput and compatibility with both stations and APs.

CONFIG_ESP32_WIFI_DYNAMIC_RX_BUFFER_NUM

Max number of WiFi dynamic RX buffers

Found in: [Component config](#) > [Wi-Fi](#)

Set the number of WiFi dynamic RX buffers, 0 means unlimited RX buffers will be allocated (provided sufficient free RAM). The size of each dynamic RX buffer depends on the size of the received data frame.

For each received data frame, the WiFi driver makes a copy to an RX buffer and then delivers it to the high layer TCP/IP stack. The dynamic RX buffer is freed after the higher layer has successfully received the data frame.

For some applications, WiFi data frames may be received faster than the application can process them. In these cases we may run out of memory if RX buffer number is unlimited (0).

If a dynamic RX buffer limit is set, it should be at least the number of static RX buffers.

CONFIG_ESP32_WIFI_TX_BUFFER

Type of WiFi TX buffers

Found in: [Component config](#) > [Wi-Fi](#)

Select type of WiFi TX buffers:

If “Static” is selected, WiFi TX buffers are allocated when WiFi is initialized and released when WiFi is de-initialized. The size of each static TX buffer is fixed to about 1.6KB.

If “Dynamic” is selected, each WiFi TX buffer is allocated as needed when a data frame is delivered to the Wifi driver from the TCP/IP stack. The buffer is freed after the data frame has been sent by the WiFi driver. The size of each dynamic TX buffer depends on the length of each data frame sent by the TCP/IP layer.

If PSRAM is enabled, “Static” should be selected to guarantee enough WiFi TX buffers. If PSRAM is disabled, “Dynamic” should be selected to improve the utilization of RAM.

Available options:

- Static (`ESP32_WIFI_STATIC_TX_BUFFER`)
- Dynamic (`ESP32_WIFI_DYNAMIC_TX_BUFFER`)

CONFIG_ESP32_WIFI_STATIC_TX_BUFFER_NUM

Max number of WiFi static TX buffers

Found in: [Component config](#) > [Wi-Fi](#)

Set the number of WiFi static TX buffers. Each buffer takes approximately 1.6KB of RAM. The static RX buffers are allocated when `esp_wifi_init()` is called, they are not released until `esp_wifi_deinit()` is called.

For each transmitted data frame from the higher layer TCP/IP stack, the WiFi driver makes a copy of it in a TX buffer. For some applications especially UDP applications, the upper layer can deliver frames faster than WiFi layer can transmit. In these cases, we may run out of TX buffers.

CONFIG_ESP32_WIFI_DYNAMIC_TX_BUFFER_NUM

Max number of WiFi dynamic TX buffers

Found in: [Component config](#) > [Wi-Fi](#)

Set the number of WiFi dynamic TX buffers. The size of each dynamic TX buffer is not fixed, it depends on the size of each transmitted data frame.

For each transmitted frame from the higher layer TCP/IP stack, the WiFi driver makes a copy of it in a TX buffer. For some applications, especially UDP applications, the upper layer can deliver frames faster than WiFi layer can transmit. In these cases, we may run out of TX buffers.

CONFIG_ESP32_WIFI_CSI_ENABLED

WiFi CSI(Channel State Information)

Found in: [Component config](#) > [Wi-Fi](#)

Select this option to enable CSI(Channel State Information) feature. CSI takes about CONFIG_ESP32_WIFI_STATIC_RX_BUFFER_NUM KB of RAM. If CSI is not used, it is better to disable this feature in order to save memory.

CONFIG_ESP32_WIFI_AMPDU_TX_ENABLED

WiFi AMPDU TX

Found in: [Component config](#) > [Wi-Fi](#)

Select this option to enable AMPDU TX feature

CONFIG_ESP32_WIFI_TX_BA_WIN

WiFi AMPDU TX BA window size

Found in: [Component config](#) > [Wi-Fi](#) > [CONFIG_ESP32_WIFI_AMPDU_TX_ENABLED](#)

Set the size of WiFi Block Ack TX window. Generally a bigger value means higher throughput but more memory. Most of time we should NOT change the default value unless special reason, e.g. test the maximum UDP TX throughput with iperf etc. For iperf test in shieldbox, the recommended value is 9~12.

CONFIG_ESP32_WIFI_AMPDU_RX_ENABLED

WiFi AMPDU RX

Found in: [Component config](#) > [Wi-Fi](#)

Select this option to enable AMPDU RX feature

CONFIG_ESP32_WIFI_RX_BA_WIN

WiFi AMPDU RX BA window size

Found in: [Component config](#) > [Wi-Fi](#) > [CONFIG_ESP32_WIFI_AMPDU_RX_ENABLED](#)

Set the size of WiFi Block Ack RX window. Generally a bigger value means higher throughput and better compatibility but more memory. Most of time we should NOT change the default value unless special reason, e.g. test the maximum UDP RX throughput with iperf etc. For iperf test in shieldbox,

the recommended value is 9~12. If PSRAM is used and WiFi memory is preferred to allocate in PSRAM first, the default and minimum value should be 16 to achieve better throughput and compatibility with both stations and APs.

CONFIG_ESP32_WIFI_NVS_ENABLED

WiFi NVS flash

Found in: [Component config](#) > [Wi-Fi](#)

Select this option to enable WiFi NVS flash

CONFIG_ESP32_WIFI_TASK_CORE_ID

WiFi Task Core ID

Found in: [Component config](#) > [Wi-Fi](#)

Pinned WiFi task to core 0 or core 1.

Available options:

- Core 0 (ESP32_WIFI_TASK_PINNED_TO_CORE_0)
- Core 1 (ESP32_WIFI_TASK_PINNED_TO_CORE_1)

CONFIG_ESP32_WIFI_SOFTAP_BEACON_MAX_LEN

Max length of WiFi SoftAP Beacon

Found in: [Component config](#) > [Wi-Fi](#)

ESP-MESH utilizes beacon frames to detect and resolve root node conflicts (see documentation). However the default length of a beacon frame can simultaneously hold only five root node identifier structures, meaning that a root node conflict of up to five nodes can be detected at one time. In the occurrence of more root nodes conflict involving more than five root nodes, the conflict resolution process will detect five of the root nodes, resolve the conflict, and re-detect more root nodes. This process will repeat until all root node conflicts are resolved. However this process can generally take a very long time.

To counter this situation, the beacon frame length can be increased such that more root nodes can be detected simultaneously. Each additional root node will require 36 bytes and should be added on top of the default beacon frame length of 752 bytes. For example, if you want to detect 10 root nodes simultaneously, you need to set the beacon frame length as 932 (752+36*5).

Setting a longer beacon length also assists with debugging as the conflicting root nodes can be identified more quickly.

CONFIG_ESP32_WIFI_MGMT_SBUF_NUM

WiFi mgmt short buffer number

Found in: [Component config](#) > [Wi-Fi](#)

Set the number of WiFi management short buffer.

CONFIG_ESP32_WIFI_DEBUG_LOG_ENABLE

Enable WiFi debug log

Found in: [Component config](#) > [Wi-Fi](#)

Select this option to enable WiFi debug log

CONFIG_ESP32_WIFI_DEBUG_LOG_LEVEL

WiFi debug log level

Found in: [Component config](#) > [Wi-Fi](#) > [CONFIG_ESP32_WIFI_DEBUG_LOG_ENABLE](#)

The WiFi log is divided into the following levels: ERROR,WARNING,INFO,DEBUG,VERBOSE. The ERROR,WARNING,INFO levels are enabled by default, and the DEBUG,VERBOSE levels can be enabled here.

Available options:

- WiFi Debug Log Debug (ESP32_WIFI_DEBUG_LOG_DEBUG)
- WiFi Debug Log Verbose (ESP32_WIFI_DEBUG_LOG_VERBOSE)

CONFIG_ESP32_WIFI_DEBUG_LOG_MODULE

WiFi debug log module

Found in: [Component config](#) > [Wi-Fi](#) > [CONFIG_ESP32_WIFI_DEBUG_LOG_ENABLE](#)

The WiFi log module contains three parts: WIFI,COEX,MESH. The WIFI module indicates the logs related to WiFi, the COEX module indicates the logs related to WiFi and BT(or BLE) coexist, the MESH module indicates the logs related to Mesh. When ESP32_WIFI_LOG_MODULE_ALL is enabled, all modules are selected.

Available options:

- WiFi Debug Log Module All (ESP32_WIFI_DEBUG_LOG_MODULE_ALL)
- WiFi Debug Log Module WiFi (ESP32_WIFI_DEBUG_LOG_MODULE_WIFI)
- WiFi Debug Log Module Coex (ESP32_WIFI_DEBUG_LOG_MODULE_COEX)
- WiFi Debug Log Module Mesh (ESP32_WIFI_DEBUG_LOG_MODULE_MESH)

CONFIG_ESP32_WIFI_DEBUG_LOG_SUBMODULE

WiFi debug log submodule

Found in: [Component config](#) > [Wi-Fi](#) > [CONFIG_ESP32_WIFI_DEBUG_LOG_ENABLE](#)

Enable this option to set the WiFi debug log submodule. Currently the log submodule contains the following parts: INIT,IOCTL,CONN,SCAN. The INIT submodule indicates the initialization process.The IOCTL submodule indicates the API calling process. The CONN submodule indicates the connecting process.The SCAN submodule indicates the scanning process.

CONFIG_ESP32_WIFI_DEBUG_LOG_SUBMODULE_ALL

WiFi Debug Log Submodule All

Found in: [Component config](#) > [Wi-Fi](#) > [CONFIG_ESP32_WIFI_DEBUG_LOG_ENABLE](#) > [CONFIG_ESP32_WIFI_DEBUG_LOG_SUBMODULE](#)

When this option is enabled, all debug submodules are selected.

CONFIG_ESP32_WIFI_DEBUG_LOG_SUBMODULE_INIT

WiFi Debug Log Submodule Init

Found in: [Component config](#) > [Wi-Fi](#) > [CONFIG_ESP32_WIFI_DEBUG_LOG_ENABLE](#) > [CONFIG_ESP32_WIFI_DEBUG_LOG_SUBMODULE](#)

CONFIG_ESP32_WIFI_DEBUG_LOG_SUBMODULE_IOCTL

WiFi Debug Log Submodule Ioctl

Found in: [Component config](#) > [Wi-Fi](#) > [CONFIG_ESP32_WIFI_DEBUG_LOG_ENABLE](#) > [CONFIG_ESP32_WIFI_DEBUG_LOG_SUBMODULE](#)

CONFIG_ESP32_WIFI_DEBUG_LOG_SUBMODULE_CONN

WiFi Debug Log Submodule Conn

Found in: Component config > Wi-Fi > CONFIG_ESP32_WIFI_DEBUG_LOG_ENABLE > CONFIG_ESP32_WIFI_DEBUG_LOG_SUBMODULE

CONFIG_ESP32_WIFI_DEBUG_LOG_SUBMODULE_SCAN

WiFi Debug Log Submodule Scan

Found in: Component config > Wi-Fi > CONFIG_ESP32_WIFI_DEBUG_LOG_ENABLE > CONFIG_ESP32_WIFI_DEBUG_LOG_SUBMODULE

CONFIG_ESP32_WIFI_IRAM_OPT

WiFi IRAM speed optimization

Found in: Component config > Wi-Fi

Select this option to place frequently called Wi-Fi library functions in IRAM. When this option is disabled, more than 10Kbytes of IRAM memory will be saved but Wi-Fi throughput will be reduced.

CONFIG_ESP32_WIFI_RX_IRAM_OPT

WiFi RX IRAM speed optimization

Found in: Component config > Wi-Fi

Select this option to place frequently called Wi-Fi library RX functions in IRAM. When this option is disabled, more than 17Kbytes of IRAM memory will be saved but Wi-Fi performance will be reduced.

CONFIG_ESP32_WIFI_ENABLE_WPA3_SAE

Enable WPA3-Personal

Found in: Component config > Wi-Fi

Select this option to allow the device to establish a WPA3-Personal connection with eligible AP's. PMF (Protected Management Frames) is a prerequisite feature for a WPA3 connection, it needs to be explicitly configured before attempting connection. Please refer to the Wi-Fi Driver API Guide for details.

PHY Contains:

- *CONFIG_ESP32_PHY_CALIBRATION_AND_DATA_STORAGE*
- *CONFIG_ESP32_PHY_INIT_DATA_IN_PARTITION*
- *CONFIG_ESP32_PHY_MAX_WIFI_TX_POWER*

CONFIG_ESP32_PHY_CALIBRATION_AND_DATA_STORAGE

Store phy calibration data in NVS

Found in: Component config > PHY

If this option is enabled, NVS will be initialized and calibration data will be loaded from there. PHY calibration will be skipped on deep sleep wakeup. If calibration data is not found, full calibration will be performed and stored in NVS. Normally, only partial calibration will be performed. If this option is disabled, full calibration will be performed.

If it's easy that your board calibrate bad data, choose 'n'. Two cases for example, you should choose 'n': 1.If your board is easy to be booted up with antenna disconnected. 2.Because of your board design, each time when you do calibration, the result are too unstable. If unsure, choose 'y'.

CONFIG_ESP32_PHY_INIT_DATA_IN_PARTITION

Use a partition to store PHY init data

Found in: [Component config](#) > [PHY](#)

If enabled, PHY init data will be loaded from a partition. When using a custom partition table, make sure that PHY data partition is included (type: 'data', subtype: 'phy'). With default partition tables, this is done automatically. If PHY init data is stored in a partition, it has to be flashed there, otherwise runtime error will occur.

If this option is not enabled, PHY init data will be embedded into the application binary.

If unsure, choose 'n'.

CONFIG_ESP32_PHY_MAX_WIFI_TX_POWER

Max WiFi TX power (dBm)

Found in: [Component config](#) > [PHY](#)

Set maximum transmit power for WiFi radio. Actual transmit power for high data rates may be lower than this setting.

Core dump Contains:

- [CONFIG_ESP32_COREDUMP_TO_FLASH_OR_UART](#)
- [CONFIG_ESP32_COREDUMP_DATA_FORMAT](#)
- [CONFIG_ESP32_COREDUMP_CHECKSUM](#)
- [CONFIG_ESP32_CORE_DUMP_MAX_TASKS_NUM](#)
- [CONFIG_ESP32_CORE_DUMP_UART_DELAY](#)
- [CONFIG_ESP32_CORE_DUMP_STACK_SIZE](#)

CONFIG_ESP32_COREDUMP_TO_FLASH_OR_UART

Data destination

Found in: [Component config](#) > [Core dump](#)

Select place to store core dump: flash, uart or none (to disable core dumps generation).

Core dumps to Flash are not available if PSRAM is used for task stacks.

If core dump is configured to be stored in flash and custom partition table is used add corresponding entry to your CSV. For examples, please see predefined partition table CSV descriptions in the `components/partition_table` directory.

Available options:

- Flash ([ESP32_ENABLE_COREDUMP_TO_FLASH](#))
- UART ([ESP32_ENABLE_COREDUMP_TO_UART](#))
- None ([ESP32_ENABLE_COREDUMP_TO_NONE](#))

CONFIG_ESP32_COREDUMP_DATA_FORMAT

Core dump data format

Found in: [Component config](#) > [Core dump](#)

Select the data format for core dump.

Available options:

- Binary format ([ESP32_COREDUMP_DATA_FORMAT_BIN](#))
- ELF format ([ESP32_COREDUMP_DATA_FORMAT_ELF](#))

CONFIG_ESP32_COREDUMP_CHECKSUM

Core dump data integrity check

Found in: *Component config > Core dump*

Select the integrity check for the core dump.

Available options:

- Use CRC32 for integrity verification (ESP32_COREDUMP_CHECKSUM_CRC32)
- Use SHA256 for integrity verification (ESP32_COREDUMP_CHECKSUM_SHA256)

CONFIG_ESP32_CORE_DUMP_MAX_TASKS_NUM

Maximum number of tasks

Found in: *Component config > Core dump*

Maximum number of tasks snapshots in core dump.

CONFIG_ESP32_CORE_DUMP_UART_DELAY

Delay before print to UART

Found in: *Component config > Core dump*

Config delay (in ms) before printing core dump to UART. Delay can be interrupted by pressing Enter key.

CONFIG_ESP32_CORE_DUMP_STACK_SIZE

Reserved stack size

Found in: *Component config > Core dump*

Size of the memory to be reserved for core dump stack. If 0 core dump process will run on the stack of crashed task/ISR, otherwise special stack will be allocated. To ensure that core dump itself will not overflow task/ISR stack set this to the value above 800. NOTE: It eats DRAM.

FAT Filesystem support

 Contains:

- *CONFIG_FATFS_CHOOSE_CODEPAGE*
- *CONFIG_FATFS_LONG_FILENAMES*
- *CONFIG_FATFS_MAX_LFN*
- *CONFIG_FATFS_API_ENCODING*
- *CONFIG_FATFS_FS_LOCK*
- *CONFIG_FATFS_TIMEOUT_MS*
- *CONFIG_FATFS_PER_FILE_CACHE*
- *CONFIG_FATFS_ALLOC_PREFER_EXTRAM*

CONFIG_FATFS_CHOOSE_CODEPAGE

OEM Code Page

Found in: *Component config > FAT Filesystem support*

OEM code page used for file name encodings.

If “Dynamic” is selected, code page can be chosen at runtime using `f_setcp` function. Note that choosing this option will increase application size by ~480kB.

Available options:

- Dynamic (all code pages supported) (FATFS_CODEPAGE_DYNAMIC)
- US (CP437) (FATFS_CODEPAGE_437)

- Arabic (CP720) (FATFS_CODEPAGE_720)
- Greek (CP737) (FATFS_CODEPAGE_737)
- KBL (CP771) (FATFS_CODEPAGE_771)
- Baltic (CP775) (FATFS_CODEPAGE_775)
- Latin 1 (CP850) (FATFS_CODEPAGE_850)
- Latin 2 (CP852) (FATFS_CODEPAGE_852)
- Cyrillic (CP855) (FATFS_CODEPAGE_855)
- Turkish (CP857) (FATFS_CODEPAGE_857)
- Portugese (CP860) (FATFS_CODEPAGE_860)
- Icelandic (CP861) (FATFS_CODEPAGE_861)
- Hebrew (CP862) (FATFS_CODEPAGE_862)
- Canadian French (CP863) (FATFS_CODEPAGE_863)
- Arabic (CP864) (FATFS_CODEPAGE_864)
- Nordic (CP865) (FATFS_CODEPAGE_865)
- Russian (CP866) (FATFS_CODEPAGE_866)
- Greek 2 (CP869) (FATFS_CODEPAGE_869)
- Japanese (DBCS) (CP932) (FATFS_CODEPAGE_932)
- Simplified Chinese (DBCS) (CP936) (FATFS_CODEPAGE_936)
- Korean (DBCS) (CP949) (FATFS_CODEPAGE_949)
- Traditional Chinese (DBCS) (CP950) (FATFS_CODEPAGE_950)

CONFIG_FATFS_LONG_FILENAMES

Long filename support

Found in: [Component config](#) > [FAT Filesystem support](#)

Support long filenames in FAT. Long filename data increases memory usage. FATFS can be configured to store the buffer for long filename data in stack or heap.

Available options:

- No long filenames (FATFS_LFN_NONE)
- Long filename buffer in heap (FATFS_LFN_HEAP)
- Long filename buffer on stack (FATFS_LFN_STACK)

CONFIG_FATFS_MAX_LFN

Max long filename length

Found in: [Component config](#) > [FAT Filesystem support](#)

Maximum long filename length. Can be reduced to save RAM.

CONFIG_FATFS_API_ENCODING

API character encoding

Found in: [Component config](#) > [FAT Filesystem support](#)

Choose encoding for character and string arguments/returns when using FATFS APIs. The encoding of arguments will usually depend on text editor settings.

Available options:

- API uses ANSI/OEM encoding (FATFS_API_ENCODING_ANSI_OEM)
- API uses UTF-16 encoding (FATFS_API_ENCODING_UTF_16)
- API uses UTF-8 encoding (FATFS_API_ENCODING_UTF_8)

CONFIG_FATFS_FS_LOCK

Number of simultaneously open files protected by lock function

Found in: [Component config](#) > [FAT Filesystem support](#)

This option sets the FATFS configuration value `_FS_LOCK`. The option `_FS_LOCK` switches file lock function to control duplicated file open and illegal operation to open objects.

* 0: Disable file lock function. To avoid volume corruption, application should avoid illegal open, remove and rename to the open objects.

* >0: Enable file lock function. The value defines how many files/sub-directories can be opened simultaneously under file lock control.

Note that the file lock control is independent of re-entrancy.

CONFIG_FATFS_TIMEOUT_MS

Timeout for acquiring a file lock, ms

Found in: [Component config](#) > [FAT Filesystem support](#)

This option sets FATFS configuration value `_FS_TIMEOUT`, scaled to milliseconds. Sets the number of milliseconds FATFS will wait to acquire a mutex when operating on an open file. For example, if one task is performing a lengthy operation, another task will wait for the first task to release the lock, and time out after amount of time set by this option.

CONFIG_FATFS_PER_FILE_CACHE

Use separate cache for each file

Found in: [Component config](#) > [FAT Filesystem support](#)

This option affects FATFS configuration value `_FS_TINY`.

If this option is set, `_FS_TINY` is 0, and each open file has its own cache, size of the cache is equal to the `_MAX_SS` variable (512 or 4096 bytes). This option uses more RAM if more than 1 file is open, but needs less reads and writes to the storage for some operations.

If this option is not set, `_FS_TINY` is 1, and single cache is used for all open files, size is also equal to `_MAX_SS` variable. This reduces the amount of heap used when multiple files are open, but increases the number of read and write operations which FATFS needs to make.

CONFIG_FATFS_ALLOC_PREFER_EXTRAM

Prefer external RAM when allocating FATFS buffers

Found in: [Component config](#) > [FAT Filesystem support](#)

When the option is enabled, internal buffers used by FATFS will be allocated from external RAM. If the allocation from external RAM fails, the buffer will be allocated from the internal RAM. Disable this option if optimizing for performance. Enable this option if optimizing for internal memory size.

Modbus configuration Contains:

- [CONFIG_FMB_COMM_MODE_RTU_EN](#)
- [CONFIG_FMB_COMM_MODE_ASCII_EN](#)
- [CONFIG_FMB_MASTER_TIMEOUT_MS_RESPOND](#)
- [CONFIG_FMB_MASTER_DELAY_MS_CONVERT](#)
- [CONFIG_FMB_QUEUE_LENGTH](#)
- [CONFIG_FMB_SERIAL_TASK_STACK_SIZE](#)
- [CONFIG_FMB_SERIAL_BUF_SIZE](#)
- [CONFIG_FMB_SERIAL_ASCII_BITS_PER_SYMB](#)
- [CONFIG_FMB_SERIAL_ASCII_TIMEOUT_RESPOND_MS](#)
- [CONFIG_FMB_SERIAL_TASK_PRIO](#)

- `CONFIG_FMB_PORT_TASK_AFFINITY`
- `CONFIG_FMB_CONTROLLER_SLAVE_ID_SUPPORT`
- `CONFIG_FMB_CONTROLLER_NOTIFY_TIMEOUT`
- `CONFIG_FMB_CONTROLLER_NOTIFY_QUEUE_SIZE`
- `CONFIG_FMB_CONTROLLER_STACK_SIZE`
- `CONFIG_FMB_EVENT_QUEUE_TIMEOUT`
- `CONFIG_FMB_TIMER_PORT_ENABLED`
- `CONFIG_FMB_TIMER_GROUP`
- `CONFIG_FMB_TIMER_INDEX`
- `CONFIG_FMB_TIMER_ISR_IN_IRAM`

CONFIG_FMB_COMM_MODE_RTU_EN

Enable Modbus stack support for RTU mode

Found in: [Component config](#) > [Modbus configuration](#)

Enable RTU Modbus communication mode option for Modbus serial stack.

CONFIG_FMB_COMM_MODE_ASCII_EN

Enable Modbus stack support for ASCII mode

Found in: [Component config](#) > [Modbus configuration](#)

Enable ASCII Modbus communication mode option for Modbus serial stack.

CONFIG_FMB_MASTER_TIMEOUT_MS_RESPOND

Slave respond timeout (Milliseconds)

Found in: [Component config](#) > [Modbus configuration](#)

If master sends a frame which is not broadcast, it has to wait sometime for slave response. if slave is not respond in this time, the master will process timeout error.

CONFIG_FMB_MASTER_DELAY_MS_CONVERT

Slave conversion delay (Milliseconds)

Found in: [Component config](#) > [Modbus configuration](#)

If master sends a broadcast frame, it has to wait conversion time to delay, then master can send next frame.

CONFIG_FMB_QUEUE_LENGTH

Modbus serial task queue length

Found in: [Component config](#) > [Modbus configuration](#)

Modbus serial driver queue length. It is used by event queue task. See the serial driver API for more information.

CONFIG_FMB_SERIAL_TASK_STACK_SIZE

Modbus serial task stack size

Found in: [Component config](#) > [Modbus configuration](#)

Modbus serial task stack size for event queue task. It may be adjusted when debugging is enabled (for example).

CONFIG_FMB_SERIAL_BUF_SIZE

Modbus serial task RX/TX buffer size

Found in: [Component config](#) > [Modbus configuration](#)

Modbus serial task RX and TX buffer size for UART driver initialization. This buffer is used for modbus frame transfer. The Modbus protocol maximum frame size is 256 bytes. Bigger size can be used for non standard implementations.

CONFIG_FMB_SERIAL_ASCII_BITS_PER_SYMB

Number of data bits per ASCII character

Found in: [Component config](#) > [Modbus configuration](#)

This option defines the number of data bits per ASCII character.

CONFIG_FMB_SERIAL_ASCII_TIMEOUT_RESPOND_MS

Response timeout for ASCII communication mode (ms)

Found in: [Component config](#) > [Modbus configuration](#)

This option defines response timeout of slave in milliseconds for ASCII communication mode. Thus the timeout will expire and allow the master program to handle the error.

CONFIG_FMB_SERIAL_TASK_PRIO

Modbus serial task priority

Found in: [Component config](#) > [Modbus configuration](#)

Modbus UART driver event task priority. The priority of Modbus controller task is equal to (CONFIG_FMB_SERIAL_TASK_PRIO - 1).

CONFIG_FMB_PORT_TASK_AFFINITY

Modbus task affinity

Found in: [Component config](#) > [Modbus configuration](#)

Allows setting the core affinity of the Modbus controller task, i.e. whether the task is pinned to particular CPU, or allowed to run on any CPU.

Available options:

- No affinity (FMB_PORT_TASK_AFFINITY_NO_AFFINITY)
- CPU0 (FMB_PORT_TASK_AFFINITY_CPU0)
- CPU1 (FMB_PORT_TASK_AFFINITY_CPU1)

CONFIG_FMB_CONTROLLER_SLAVE_ID_SUPPORT

Modbus controller slave ID support

Found in: [Component config](#) > [Modbus configuration](#)

Modbus slave ID support enable. When enabled the Modbus <Report Slave ID> command is supported by stack.

CONFIG_FMB_CONTROLLER_SLAVE_ID

Modbus controller slave ID

Found in: Component config > Modbus configuration > CONFIG_FMB_CONTROLLER_SLAVE_ID_SUPPORT

Modbus slave ID value to identify modbus device in the network using <Report Slave ID> command. Most significant byte of ID is used as short device ID and other three bytes used as long ID.

CONFIG_FMB_CONTROLLER_NOTIFY_TIMEOUT

Modbus controller notification timeout (ms)

Found in: Component config > Modbus configuration

Modbus controller notification timeout in milliseconds. This timeout is used to send notification about accessed parameters.

CONFIG_FMB_CONTROLLER_NOTIFY_QUEUE_SIZE

Modbus controller notification queue size

Found in: Component config > Modbus configuration

Modbus controller notification queue size. The notification queue is used to get information about accessed parameters.

CONFIG_FMB_CONTROLLER_STACK_SIZE

Modbus controller stack size

Found in: Component config > Modbus configuration

Modbus controller task stack size. The Stack size may be adjusted when debug mode is used which requires more stack size (for example).

CONFIG_FMB_EVENT_QUEUE_TIMEOUT

Modbus stack event queue timeout (ms)

Found in: Component config > Modbus configuration

Modbus stack event queue timeout in milliseconds. This may help to optimize Modbus stack event processing time.

CONFIG_FMB_TIMER_PORT_ENABLED

Modbus slave stack use timer for 3.5T symbol time measurement

Found in: Component config > Modbus configuration

If this option is set the Modbus stack uses timer for T3.5 time measurement. Else the internal UART TOUT timeout is used for 3.5T symbol time measurement.

CONFIG_FMB_TIMER_GROUP

Modbus Timer group number

Found in: Component config > Modbus configuration

Modbus Timer group number that is used for timeout measurement.

CONFIG_FMB_TIMER_INDEX

Modbus Timer index in the group

Found in: [Component config](#) > [Modbus configuration](#)

Modbus Timer Index in the group that is used for timeout measurement.

CONFIG_FMB_TIMER_ISR_IN_IRAM

Place timer interrupt handler into IRAM

Found in: [Component config](#) > [Modbus configuration](#)

This option places Modbus timer IRQ handler into IRAM. This allows to avoid delays related to processing of non-IRAM-safe interrupts during a flash write operation (NVS updating a value, or some other flash API which has to perform a read/write operation and disable CPU cache). This option has dependency with the `UART_ISR_IN_IRAM` option which places UART interrupt handler into IRAM to prevent delays related to processing of UART events.

FreeRTOS Contains:

- `CONFIG_FREERTOS_UNICORE`
- `CONFIG_FREERTOS_CORETIMER`
- `CONFIG_FREERTOS_HZ`
- `CONFIG_FREERTOS_ASSERT_ON_UNTESTED_FUNCTION`
- `CONFIG_FREERTOS_CHECK_STACKOVERFLOW`
- `CONFIG_FREERTOS_WATCHPOINT_END_OF_STACK`
- `CONFIG_FREERTOS_INTERRUPT_BACKTRACE`
- `CONFIG_FREERTOS_THREAD_LOCAL_STORAGE_POINTERS`
- `CONFIG_FREERTOS_ASSERT`
- `CONFIG_FREERTOS_IDLE_TASK_STACKSIZE`
- `CONFIG_FREERTOS_ISR_STACKSIZE`
- `CONFIG_FREERTOS_LEGACY_HOOKS`
- `CONFIG_FREERTOS_MAX_TASK_NAME_LEN`
- `CONFIG_FREERTOS_SUPPORT_STATIC_ALLOCATION`
- `CONFIG_FREERTOS_TIMER_TASK_PRIORITY`
- `CONFIG_FREERTOS_TIMER_TASK_STACK_DEPTH`
- `CONFIG_FREERTOS_TIMER_QUEUE_LENGTH`
- `CONFIG_FREERTOS_QUEUE_REGISTRY_SIZE`
- `CONFIG_FREERTOS_USE_TRACE_FACILITY`
- `CONFIG_FREERTOS_GENERATE_RUN_TIME_STATS`
- `CONFIG_FREERTOS_USE_TICKLESS_IDLE`
- `CONFIG_FREERTOS_DEBUG_INTERNALS`
- `CONFIG_FREERTOS_TASK_FUNCTION_WRAPPER`
- `CONFIG_FREERTOS_CHECK_MUTEX_GIVEN_BY_OWNER`
- `CONFIG_FREERTOS_CHECK_PORT_CRITICAL_COMPLIANCE`

CONFIG_FREERTOS_UNICORE

Run FreeRTOS only on first core

Found in: [Component config](#) > [FreeRTOS](#)

This version of FreeRTOS normally takes control of all cores of the CPU. Select this if you only want to start it on the first core. This is needed when e.g. another process needs complete control over the second core.

This invisible config value sets the value of `tskNO_AFFINITY` in `task.h`. # Intended to be used as a constant from other Kconfig files. # Value is (32-bit) `INT_MAX`.

CONFIG_FREERTOS_CORETIMER

Xtensa timer to use as the FreeRTOS tick source

Found in: [Component config > FreeRTOS](#)

FreeRTOS needs a timer with an associated interrupt to use as the main tick source to increase counters, run timers and do pre-emptive multitasking with. There are multiple timers available to do this, with different interrupt priorities. Check

Available options:

- Timer 0 (int 6, level 1) (FREERTOS_CORETIMER_0)
Select this to use timer 0
- Timer 1 (int 15, level 3) (FREERTOS_CORETIMER_1)
Select this to use timer 1

CONFIG_FREERTOS_HZ

Tick rate (Hz)

Found in: [Component config > FreeRTOS](#)

Select the tick rate at which FreeRTOS does pre-emptive context switching.

CONFIG_FREERTOS_ASSERT_ON_UNTESTED_FUNCTION

Halt when an SMP-untested function is called

Found in: [Component config > FreeRTOS](#)

Some functions in FreeRTOS have not been thoroughly tested yet when moving to the SMP implementation of FreeRTOS. When this option is enabled, these functions will throw an assert().

CONFIG_FREERTOS_CHECK_STACKOVERFLOW

Check for stack overflow

Found in: [Component config > FreeRTOS](#)

FreeRTOS can check for stack overflows in threads and trigger an user function called `vApplicationStackOverflowHook` when this happens.

Available options:

- No checking (FREERTOS_CHECK_STACKOVERFLOW_NONE)
Do not check for stack overflows (configCHECK_FOR_STACK_OVERFLOW=0)
- Check by stack pointer value (FREERTOS_CHECK_STACKOVERFLOW_PTRVAL)
Check for stack overflows on each context switch by checking if the stack pointer is in a valid range. Quick but does not detect stack overflows that happened between context switches (configCHECK_FOR_STACK_OVERFLOW=1)
- Check using canary bytes (FREERTOS_CHECK_STACKOVERFLOW_CANARY)
Places some magic bytes at the end of the stack area and on each context switch, check if these bytes are still intact. More thorough than just checking the pointer, but also slightly slower. (configCHECK_FOR_STACK_OVERFLOW=2)

CONFIG_FREERTOS_WATCHPOINT_END_OF_STACK

Set a debug watchpoint as a stack overflow check

Found in: [Component config > FreeRTOS](#)

FreeRTOS can check if a stack has overflowed its bounds by checking either the value of the stack pointer or by checking the integrity of canary bytes. (See `FREERTOS_CHECK_STACKOVERFLOW` for more information.) These checks only happen on a context switch, and the situation that caused the stack overflow may already be long gone by then. This option will use the debug memory watchpoint 1

(the second one) to allow breaking into the debugger (or panic'ing) as soon as any of the last 32 bytes on the stack of a task are overwritten. The side effect is that using gdb, you effectively only have one watchpoint; the 2nd one is overwritten as soon as a task switch happens.

This check only triggers if the stack overflow writes within 4 bytes of the end of the stack, rather than overshooting further, so it is worth combining this approach with one of the other stack overflow check methods.

When this watchpoint is hit, gdb will stop with a SIGTRAP message. When no JTAG OCD is attached, esp-idf will panic on an unhandled debug exception.

CONFIG_FREERTOS_INTERRUPT_BACKTRACE

Enable backtrace from interrupt to task context

Found in: [Component config > FreeRTOS](#)

If this option is enabled, interrupt stack frame will be modified to point to the code of the interrupted task as its return address. This helps the debugger (or the panic handler) show a backtrace from the interrupt to the task which was interrupted. This also works for nested interrupts: higher level interrupt stack can be traced back to the lower level interrupt. This option adds 4 instructions to the interrupt dispatching code.

CONFIG_FREERTOS_THREAD_LOCAL_STORAGE_POINTERS

Number of thread local storage pointers

Found in: [Component config > FreeRTOS](#)

FreeRTOS has the ability to store per-thread pointers in the task control block. This controls the number of pointers available.

This value must be at least 1. Index 0 is reserved for use by the pthreads API thread-local-storage. Other indexes can be used for any desired purpose.

CONFIG_FREERTOS_ASSERT

FreeRTOS assertions

Found in: [Component config > FreeRTOS](#)

Failed FreeRTOS configASSERT() assertions can be configured to behave in different ways.

Available options:

- abort() on failed assertions (FREERTOS_ASSERT_FAIL_ABORT)
If a FreeRTOS configASSERT() fails, FreeRTOS will abort() and halt execution. The panic handler can be configured to handle the outcome of an abort() in different ways.
- Print and continue failed assertions (FREERTOS_ASSERT_FAIL_PRINT_CONTINUE)
If a FreeRTOS assertion fails, print it out and continue.
- Disable FreeRTOS assertions (FREERTOS_ASSERT_DISABLE)
FreeRTOS configASSERT() will not be compiled into the binary.

CONFIG_FREERTOS_IDLE_TASK_STACKSIZE

Idle Task stack size

Found in: [Component config > FreeRTOS](#)

The idle task has its own stack, sized in bytes. The default size is enough for most uses. Size can be reduced to 768 bytes if no (or simple) FreeRTOS idle hooks are used and pthread local storage or FreeRTOS local storage cleanup callbacks are not used.

The stack size may need to be increased above the default if the app installs idle or thread local storage cleanup hooks that use a lot of stack memory.

CONFIG_FREERTOS_ISR_STACKSIZE

ISR stack size

Found in: [Component config](#) > [FreeRTOS](#)

The interrupt handlers have their own stack. The size of the stack can be defined here. Each processor has its own stack, so the total size occupied will be twice this.

CONFIG_FREERTOS_LEGACY_HOOKS

Use FreeRTOS legacy hooks

Found in: [Component config](#) > [FreeRTOS](#)

FreeRTOS offers a number of hooks/callback functions that are called when a timer tick happens, the idle thread runs etc. esp-idf replaces these by runtime registerable hooks using the `esp_register_freertos_xxx_hook` system, but for legacy reasons the old hooks can also still be enabled. Please enable this only if you have code that for some reason can't be migrated to the `esp_register_freertos_xxx_hook` system.

CONFIG_FREERTOS_MAX_TASK_NAME_LEN

Maximum task name length

Found in: [Component config](#) > [FreeRTOS](#)

Changes the maximum task name length. Each task allocated will include this many bytes for a task name. Using a shorter value saves a small amount of RAM, a longer value allows more complex names.

For most uses, the default of 16 is OK.

CONFIG_FREERTOS_SUPPORT_STATIC_ALLOCATION

Enable FreeRTOS static allocation API

Found in: [Component config](#) > [FreeRTOS](#)

FreeRTOS gives the application writer the ability to instead provide the memory themselves, allowing the following objects to optionally be created without any memory being allocated dynamically:

- Tasks
- Software Timers (Daemon task is still dynamic. See documentation)
- Queues
- Event Groups
- Binary Semaphores
- Counting Semaphores
- Recursive Semaphores
- Mutexes

Whether it is preferable to use static or dynamic memory allocation is dependent on the application, and the preference of the application writer. Both methods have pros and cons, and both methods can be used within the same RTOS application.

Creating RTOS objects using statically allocated RAM has the benefit of providing the application writer with more control: RTOS objects can be placed at specific memory locations. The maximum RAM footprint can be determined at link time, rather than run time. The application writer does not need to concern themselves with graceful handling of memory allocation failures. It allows the RTOS to be used in applications that simply don't allow any dynamic memory allocation (although FreeRTOS includes allocation schemes that can overcome most objections).

CONFIG_FREERTOS_ENABLE_STATIC_TASK_CLEAN_UP

Enable static task clean up hook

Found in: [Component config](#) > [FreeRTOS](#) > [CONFIG_FREERTOS_SUPPORT_STATIC_ALLOCATION](#)

Enable this option to make FreeRTOS call the static task clean up hook when a task is deleted.

Bear in mind that if this option is enabled you will need to implement the following function:

```
void vPortCleanUpTCB ( void \*pxTCB ) {  
    // place clean up code here  
}
```

CONFIG_FREERTOS_TIMER_TASK_PRIORITY

FreeRTOS timer task priority

Found in: [Component config](#) > [FreeRTOS](#)

The timer service task (primarily) makes use of existing FreeRTOS features, allowing timer functionality to be added to an application with minimal impact on the size of the application's executable binary.

Use this constant to define the priority that the timer task will run at.

CONFIG_FREERTOS_TIMER_TASK_STACK_DEPTH

FreeRTOS timer task stack size

Found in: [Component config](#) > [FreeRTOS](#)

The timer service task (primarily) makes use of existing FreeRTOS features, allowing timer functionality to be added to an application with minimal impact on the size of the application's executable binary.

Use this constant to define the size (in bytes) of the stack allocated for the timer task.

CONFIG_FREERTOS_TIMER_QUEUE_LENGTH

FreeRTOS timer queue length

Found in: [Component config](#) > [FreeRTOS](#)

FreeRTOS provides a set of timer related API functions. Many of these functions use a standard FreeRTOS queue to send commands to the timer service task. The queue used for this purpose is called the 'timer command queue'. The 'timer command queue' is private to the FreeRTOS timer implementation, and cannot be accessed directly.

For most uses the default value of 10 is OK.

CONFIG_FREERTOS_QUEUE_REGISTRY_SIZE

FreeRTOS queue registry size

Found in: [Component config](#) > [FreeRTOS](#)

FreeRTOS uses the queue registry as a means for kernel aware debuggers to locate queues, semaphores, and mutexes. The registry allows for a textual name to be associated with a queue for easy identification within a debugging GUI. A value of 0 will disable queue registry functionality, and a value larger than 0 will specify the number of queues/semaphores/mutexes that the registry can hold.

CONFIG_FREERTOS_USE_TRACE_FACILITY

Enable FreeRTOS trace facility

Found in: [Component config](#) > [FreeRTOS](#)

If enabled, configUSE_TRACE_FACILITY will be defined as 1 in FreeRTOS. This will allow the usage of trace facility functions such as uxTaskGetSystemState().

CONFIG_FREERTOS_USE_STATS_FORMATTING_FUNCTIONS

Enable FreeRTOS stats formatting functions

Found in: [Component config](#) > [FreeRTOS](#) > [CONFIG_FREERTOS_USE_TRACE_FACILITY](#)

If enabled, configUSE_STATS_FORMATTING_FUNCTIONS will be defined as 1 in FreeRTOS. This will allow the usage of stats formatting functions such as vTaskList().

CONFIG_FREERTOS_VTASKLIST_INCLUDE_COREID

Enable display of xCoreID in vTaskList

Found in: [Component config](#) > [FreeRTOS](#) > [CONFIG_FREERTOS_USE_TRACE_FACILITY](#) > [CONFIG_FREERTOS_USE_STATS_FORMATTING_FUNCTIONS](#)

If enabled, this will include an extra column when vTaskList is called to display the CoreID the task is pinned to (0,1) or -1 if not pinned.

CONFIG_FREERTOS_GENERATE_RUN_TIME_STATS

Enable FreeRTOS to collect run time stats

Found in: [Component config](#) > [FreeRTOS](#)

If enabled, configGENERATE_RUN_TIME_STATS will be defined as 1 in FreeRTOS. This will allow FreeRTOS to collect information regarding the usage of processor time amongst FreeRTOS tasks. Run time stats are generated using either the ESP Timer or the CPU Clock as the clock source (Note that run time stats are only valid until the clock source overflows). The function vTaskGetRunTimeStats() will also be available if FREERTOS_USE_STATS_FORMATTING_FUNCTIONS and FREERTOS_USE_TRACE_FACILITY are enabled. vTaskGetRunTimeStats() will display the run time of each task as a % of the total run time of all CPUs (task run time / no of CPUs) / (total run time / 100)

CONFIG_FREERTOS_RUN_TIME_STATS_CLK

Choose the clock source for run time stats

Found in: [Component config](#) > [FreeRTOS](#) > [CONFIG_FREERTOS_GENERATE_RUN_TIME_STATS](#)

Choose the clock source for FreeRTOS run time stats. Options are CPU0's CPU Clock or the ESP Timer. Both clock sources are 32 bits. The CPU Clock can run at a higher frequency hence provide a finer resolution but will overflow much quicker. Note that run time stats are only valid until the clock source overflows.

Available options:

- Use ESP TIMER for run time stats (FREERTOS_RUN_TIME_STATS_USING_ESP_TIMER)
ESP Timer will be used as the clock source for FreeRTOS run time stats. The ESP Timer runs at a frequency of 1MHz regardless of Dynamic Frequency Scaling. Therefore the ESP Timer will overflow in approximately 4290 seconds.
- Use CPU Clock for run time stats (FREERTOS_RUN_TIME_STATS_USING_CPU_CLK)
CPU Clock will be used as the clock source for the generation of run time stats. The CPU Clock has a frequency dependent on ESP32_DEFAULT_CPU_FREQ_MHZ and Dynamic Frequency Scaling (DFS). Therefore the CPU Clock frequency can fluctuate between 80 to 240MHz. Run time stats generated using the CPU Clock represents the number of CPU

cycles each task is allocated and DOES NOT reflect the amount of time each task runs for (as CPU clock frequency can change). If the CPU clock consistently runs at the maximum frequency of 240MHz, it will overflow in approximately 17 seconds.

CONFIG_FREERTOS_USE_TICKLESS_IDLE

Tickless idle support

Found in: [Component config](#) > [FreeRTOS](#)

If power management support is enabled, FreeRTOS will be able to put the system into light sleep mode when no tasks need to run for a number of ticks. This number can be set using FREERTOS_IDLE_TIME_BEFORE_SLEEP option. This feature is also known as “automatic light sleep”

Note that timers created using esp_timer APIs may prevent the system from entering sleep mode, even when no tasks need to run.

If disabled, automatic light sleep support will be disabled.

CONFIG_FREERTOS_IDLE_TIME_BEFORE_SLEEP

Minimum number of ticks to enter sleep mode for

Found in: [Component config](#) > [FreeRTOS](#) > [CONFIG_FREERTOS_USE_TICKLESS_IDLE](#)

FreeRTOS will enter light sleep mode if no tasks need to run for this number of ticks.

CONFIG_FREERTOS_DEBUG_INTERNALS

Debug FreeRTOS internals

Found in: [Component config](#) > [FreeRTOS](#)

Enable this option to show the menu with internal FreeRTOS debugging features. This option does not change any code by itself, it just shows/hides some options.

Contains:

- [CONFIG_FREERTOS_PORTMUX_DEBUG](#)
- [CONFIG_FREERTOS_PORTMUX_DEBUG_RECURSIVE](#)

CONFIG_FREERTOS_PORTMUX_DEBUG

Debug portMUX portENTER_CRITICAL/portEXIT_CRITICAL

Found in: [Component config](#) > [FreeRTOS](#) > [CONFIG_FREERTOS_DEBUG_INTERNALS](#)

If enabled, debug information (including integrity checks) will be printed to UART for the port-specific MUX implementation.

CONFIG_FREERTOS_PORTMUX_DEBUG_RECURSIVE

Debug portMUX Recursion

Found in: [Component config](#) > [FreeRTOS](#) > [CONFIG_FREERTOS_DEBUG_INTERNALS](#)

If enabled, additional debug information will be printed for recursive portMUX usage.

CONFIG_FREERTOS_TASK_FUNCTION_WRAPPER

Enclose all task functions in a wrapper function

Found in: [Component config > FreeRTOS](#)

If enabled, all FreeRTOS task functions will be enclosed in a wrapper function. If a task function mistakenly returns (i.e. does not delete), the call flow will return to the wrapper function. The wrapper function will then log an error and abort the application. This option is also required for GDB backtraces and C++ exceptions to work correctly inside top-level task functions.

CONFIG_FREERTOS_CHECK_MUTEX_GIVEN_BY_OWNER

Check that mutex semaphore is given by owner task

Found in: [Component config > FreeRTOS](#)

If enabled, assert that when a mutex semaphore is given, the task giving the semaphore is the task which is currently holding the mutex.

CONFIG_FREERTOS_CHECK_PORT_CRITICAL_COMPLIANCE

Tests compliance with Vanilla FreeRTOS port*_CRITICAL calls

Found in: [Component config > FreeRTOS](#)

If enabled, context of port*_CRITICAL calls (ISR or Non-ISR) would be checked to be in compliance with Vanilla FreeRTOS. e.g Calling port*_CRITICAL from ISR context would cause assert failure

Heap memory debugging Contains:

- [CONFIG_HEAP_CORRUPTION_DETECTION](#)
- [CONFIG_HEAP_TRACING_DEST](#)
- [CONFIG_HEAP_TRACING_STACK_DEPTH](#)
- [CONFIG_HEAP_TASK_TRACKING](#)

CONFIG_HEAP_CORRUPTION_DETECTION

Heap corruption detection

Found in: [Component config > Heap memory debugging](#)

Enable heap poisoning features to detect heap corruption caused by out-of-bounds access to heap memory.

See the “Heap Memory Debugging” page of the IDF documentation for a description of each level of heap corruption detection.

Available options:

- Basic (no poisoning) (HEAP_POISONING_DISABLED)
- Light impact (HEAP_POISONING_LIGHT)
- Comprehensive (HEAP_POISONING_COMPREHENSIVE)

CONFIG_HEAP_TRACING_DEST

Heap tracing

Found in: [Component config > Heap memory debugging](#)

Enables the heap tracing API defined in esp_heap_trace.h.

This function causes a moderate increase in IRAM code size and a minor increase in heap function (malloc/free/realloc) CPU overhead, even when the tracing feature is not used. So it's best to keep it disabled unless tracing is being used.

Available options:

- Disabled (HEAP_TRACING_OFF)
- Standalone (HEAP_TRACING_STANDALONE)
- Host-based (HEAP_TRACING_TOHOST)

CONFIG_HEAP_TRACING_STACK_DEPTH

Heap tracing stack depth

Found in: [Component config](#) > [Heap memory debugging](#)

Number of stack frames to save when tracing heap operation callers.

More stack frames uses more memory in the heap trace buffer (and slows down allocation), but can provide useful information.

CONFIG_HEAP_TASK_TRACKING

Enable heap task tracking

Found in: [Component config](#) > [Heap memory debugging](#)

Enables tracking the task responsible for each heap allocation.

This function depends on heap poisoning being enabled and adds four more bytes of overhead for each block allocated.

jsmn Contains:

- [CONFIG_JSMN_PARENT_LINKS](#)
- [CONFIG_JSMN_STRICT](#)

CONFIG_JSMN_PARENT_LINKS

Enable parent links

Found in: [Component config](#) > [jsmn](#)

You can access to parent node of parsed json

CONFIG_JSMN_STRICT

Enable strict mode

Found in: [Component config](#) > [jsmn](#)

In strict mode primitives are: numbers and booleans

libsodium Contains:

- [CONFIG_LIBSODIUM_USE_MBEDTLS_SHA](#)

CONFIG_LIBSODIUM_USE_MBEDTLS_SHA

Use mbedTLS SHA256 & SHA512 implementations

Found in: [Component config](#) > [libsodium](#)

If this option is enabled, libsodium will use thin wrappers around mbedTLS for SHA256 & SHA512 operations.

This saves some code size if mbedTLS is also used. However it is incompatible with hardware SHA acceleration (due to the way libsodium's API manages SHA state).

Log output Contains:

- [CONFIG_LOG_DEFAULT_LEVEL](#)
- [CONFIG_LOG_COLORS](#)
- [CONFIG_LOG_TIMESTAMP_SOURCE](#)

CONFIG_LOG_DEFAULT_LEVEL

Default log verbosity

Found in: [Component config](#) > [Log output](#)

Specify how much output to see in logs by default. You can set lower verbosity level at runtime using `esp_log_level_set` function.

Note that this setting limits which log statements are compiled into the program. So setting this to, say, “Warning” would mean that changing log level to “Debug” at runtime will not be possible.

Available options:

- No output (`LOG_DEFAULT_LEVEL_NONE`)
- Error (`LOG_DEFAULT_LEVEL_ERROR`)
- Warning (`LOG_DEFAULT_LEVEL_WARN`)
- Info (`LOG_DEFAULT_LEVEL_INFO`)
- Debug (`LOG_DEFAULT_LEVEL_DEBUG`)
- Verbose (`LOG_DEFAULT_LEVEL_VERBOSE`)

CONFIG_LOG_COLORS

Use ANSI terminal colors in log output

Found in: [Component config](#) > [Log output](#)

Enable ANSI terminal color codes in bootloader output.

In order to view these, your terminal program must support ANSI color codes.

CONFIG_LOG_TIMESTAMP_SOURCE

Log Timestamps

Found in: [Component config](#) > [Log output](#)

Choose what sort of timestamp is displayed in the log output:

- Milliseconds since boot is calculated from the RTOS tick count multiplied by the tick period. This time will reset after a software reboot. e.g. (90000)
- System time is taken from POSIX time functions which use the ESP32’s RTC and FRC1 timers to maintain an accurate time. The system time is initialized to 0 on startup, it can be set with an SNTP sync, or with POSIX time functions. This time will not reset after a software reboot. e.g. (00:01:30.000)
- NOTE: Currently this will not get used in logging from binary blobs (i.e WiFi & Bluetooth libraries), these will always print milliseconds since boot.

Available options:

- Milliseconds Since Boot (`LOG_TIMESTAMP_SOURCE_RTOS`)
- System Time (`LOG_TIMESTAMP_SOURCE_SYSTEM`)

LWIP Contains:

- [CONFIG_LWIP_LOCAL_HOSTNAME](#)
- [CONFIG_LWIP_DNS_SUPPORT_MDNS_QUERIES](#)
- [CONFIG_LWIP_L2_TO_L3_COPY](#)
- [CONFIG_LWIP_IRAM_OPTIMIZATION](#)

- `CONFIG_LWIP_TIMERS_ONDEMAND`
- `CONFIG_LWIP_MAX_SOCKETS`
- `CONFIG_LWIP_USE_ONLY_LWIP_SELECT`
- `CONFIG_LWIP_SO_LINGER`
- `CONFIG_LWIP_SO_REUSE`
- `CONFIG_LWIP_SO_RCVBUF`
- `CONFIG_LWIP_NETBUF_RECVINFO`
- `CONFIG_LWIP_IP4_FRAG`
- `CONFIG_LWIP_IP6_FRAG`
- `CONFIG_LWIP_IP4_REASSEMBLY`
- `CONFIG_LWIP_IP6_REASSEMBLY`
- `CONFIG_LWIP_STATS`
- `CONFIG_LWIP_ETHARP_TRUST_IP_MAC`
- `CONFIG_LWIP_ESP_GRATUITOUS_ARP`
- `CONFIG_LWIP_TCPIP_RECVMBOX_SIZE`
- `CONFIG_LWIP_DHCP_DOES_ARP_CHECK`
- `CONFIG_LWIP_DHCP_DISABLE_CLIENT_ID`
- `CONFIG_LWIP_DHCP_RESTORE_LAST_IP`
- `DHCP server`
- `CONFIG_LWIP_AUTOIP`
- `CONFIG_LWIP_IPV6_AUTOCONFIG`
- `CONFIG_LWIP_NETIF_LOOPBACK`
- `TCP`
- `UDP`
- `CONFIG_LWIP_TCPIP_TASK_STACK_SIZE`
- `CONFIG_LWIP_TCPIP_TASK_AFFINITY`
- `CONFIG_LWIP_PPP_SUPPORT`
- `CONFIG_LWIP_IPV6_MEMP_NUM_ND6_QUEUE`
- `CONFIG_LWIP_IPV6_ND6_NUM_NEIGHBORS`
- `CONFIG_LWIP_PPP_NOTIFY_PHASE_SUPPORT`
- `CONFIG_LWIP_PPP_PAP_SUPPORT`
- `CONFIG_LWIP_PPP_CHAP_SUPPORT`
- `CONFIG_LWIP_PPP_MSCHAP_SUPPORT`
- `CONFIG_LWIP_PPP_MPPE_SUPPORT`
- `CONFIG_LWIP_PPP_DEBUG_ON`
- `ICMP`
- `LWIP RAW API`
- `SNTP`

CONFIG_LWIP_LOCAL_HOSTNAME

Local netif hostname

Found in: `Component config > LWIP`

The default name this device will report to other devices on the network. Could be updated at runtime with `esp_netif_set_hostname()`

CONFIG_LWIP_DNS_SUPPORT_MDNS_QUERIES

Enable mDNS queries in resolving host name

Found in: `Component config > LWIP`

If this feature is enabled, standard API such as `gethostbyname` support .local addresses by sending one shot multicast mDNS query

CONFIG_LWIP_L2_TO_L3_COPY

Enable copy between Layer2 and Layer3 packets

Found in: [Component config](#) > [LWIP](#)

If this feature is enabled, all traffic from layer2(WIFI Driver) will be copied to a new buffer before sending it to layer3(LWIP stack), freeing the layer2 buffer. Please be notified that the total layer2 receiving buffer is fixed and ESP32 currently supports 25 layer2 receiving buffer, when layer2 buffer runs out of memory, then the incoming packets will be dropped in hardware. The layer3 buffer is allocated from the heap, so the total layer3 receiving buffer depends on the available heap size, when heap runs out of memory, no copy will be sent to layer3 and packet will be dropped in layer2. Please make sure you fully understand the impact of this feature before enabling it.

CONFIG_LWIP_IRAM_OPTIMIZATION

Enable LWIP IRAM optimization

Found in: [Component config](#) > [LWIP](#)

If this feature is enabled, some functions relating to RX/TX in LWIP will be put into IRAM, it can improve UDP/TCP throughput by >10% for single core mode, it doesn't help too much for dual core mode. On the other hand, it needs about 10KB IRAM for these optimizations.

If this feature is disabled, all lwip functions will be put into FLASH.

CONFIG_LWIP_TIMERS_ONDEMAND

Enable LWIP Timers on demand

Found in: [Component config](#) > [LWIP](#)

If this feature is enabled, IGMP and MLD6 timers will be activated only when joining groups or receiving QUERY packets.

This feature will reduce the power consumption for applications which do not use IGMP and MLD6.

CONFIG_LWIP_MAX_SOCKETS

Max number of open sockets

Found in: [Component config](#) > [LWIP](#)

Sockets take up a certain amount of memory, and allowing fewer sockets to be open at the same time conserves memory. Specify the maximum amount of sockets here. The valid value is from 1 to 16.

CONFIG_LWIP_USE_ONLY_LWIP_SELECT

Support LWIP socket select() only

Found in: [Component config](#) > [LWIP](#)

The virtual filesystem layer of select() redirects sockets to lwip_select() and non-socket file descriptors to their respective driver implementations. If this option is enabled then all calls of select() will be redirected to lwip_select(), therefore, select can be used for sockets only.

CONFIG_LWIP_SO_LINGER

Enable SO_LINGER processing

Found in: [Component config](#) > [LWIP](#)

Enabling this option allows SO_LINGER processing. l_onoff = 1,l_linger can set the timeout.

If l_linger=0, When a connection is closed, TCP will terminate the connection. This means that TCP will discard any data packets stored in the socket send buffer and send an RST to the peer.

If `l_linger!=0`, Then `closesocket()` calls to block the process until the remaining data packets has been sent or timed out.

CONFIG_LWIP_SO_REUSE

Enable `SO_REUSEADDR` option

Found in: [Component config](#) > [LWIP](#)

Enabling this option allows binding to a port which remains in `TIME_WAIT`.

CONFIG_LWIP_SO_REUSE_RXTOALL

`SO_REUSEADDR` copies broadcast/multicast to all matches

Found in: [Component config](#) > [LWIP](#) > [CONFIG_LWIP_SO_REUSE](#)

Enabling this option means that any incoming broadcast or multicast packet will be copied to all of the local sockets that it matches (may be more than one if `SO_REUSEADDR` is set on the socket.)

This increases memory overhead as the packets need to be copied, however they are only copied per matching socket. You can safely disable it if you don't plan to receive broadcast or multicast traffic on more than one socket at a time.

CONFIG_LWIP_SO_RCVBUF

Enable `SO_RCVBUF` option

Found in: [Component config](#) > [LWIP](#)

Enabling this option allows checking for available data on a netconn.

CONFIG_LWIP_NETBUF_RECVINFO

Enable `IP_PKTINFO` option

Found in: [Component config](#) > [LWIP](#)

Enabling this option allows checking for the destination address of a received IPv4 Packet.

CONFIG_LWIP_IP4_FRAG

Enable fragment outgoing IP4 packets

Found in: [Component config](#) > [LWIP](#)

Enabling this option allows fragmenting outgoing IP4 packets if their size exceeds MTU.

CONFIG_LWIP_IP6_FRAG

Enable fragment outgoing IP6 packets

Found in: [Component config](#) > [LWIP](#)

Enabling this option allows fragmenting outgoing IP6 packets if their size exceeds MTU.

CONFIG_LWIP_IP4_REASSEMBLY

Enable reassembly incoming fragmented IP4 packets

Found in: [Component config](#) > [LWIP](#)

Enabling this option allows reassembling incoming fragmented IP4 packets.

CONFIG_LWIP_IP6_REASSEMBLY

Enable reassembly incoming fragmented IP6 packets

Found in: [Component config](#) > [LWIP](#)

Enabling this option allows reassembling incoming fragmented IP6 packets.

CONFIG_LWIP_STATS

Enable LWIP statistics

Found in: [Component config](#) > [LWIP](#)

Enabling this option allows LWIP statistics

CONFIG_LWIP_ETHARP_TRUST_IP_MAC

Enable LWIP ARP trust

Found in: [Component config](#) > [LWIP](#)

Enabling this option allows ARP table to be updated.

If this option is enabled, the incoming IP packets cause the ARP table to be updated with the source MAC and IP addresses supplied in the packet. You may want to disable this if you do not trust LAN peers to have the correct addresses, or as a limited approach to attempt to handle spoofing. If disabled, lwIP will need to make a new ARP request if the peer is not already in the ARP table, adding a little latency. The peer **is** in the ARP table if it requested our address before. Also notice that this slows down input processing of every IP packet!

There are two known issues in real application if this feature is enabled: - The LAN peer may have bug to update the ARP table after the ARP entry is aged out. If the ARP entry on the LAN peer is aged out but failed to be updated, all IP packets sent from LWIP to the LAN peer will be dropped by LAN peer. - The LAN peer may not be trustful, the LAN peer may send IP packets to LWIP with two different MACs, but the same IP address. If this happens, the LWIP has problem to receive IP packets from LAN peer.

So the recommendation is to disable this option. Here the LAN peer means the other side to which the ESP station or soft-AP is connected.

CONFIG_LWIP_ESP_GRATUITOUS_ARP

Send gratuitous ARP periodically

Found in: [Component config](#) > [LWIP](#)

Enable this option allows to send gratuitous ARP periodically.

This option solve the compatibility issues.If the ARP table of the AP is old, and the AP doesn't send ARP request to update it's ARP table, this will lead to the STA sending IP packet fail. Thus we send gratuitous ARP periodically to let AP update it's ARP table.

CONFIG_LWIP_GARP_TMR_INTERVAL

GARP timer interval(seconds)

Found in: [Component config](#) > [LWIP](#) > [CONFIG_LWIP_ESP_GRATUITOUS_ARP](#)

Set the timer interval for gratuitous ARP. The default value is 60s

CONFIG_LWIP_TCPIP_RECVMBOX_SIZE

TCPIP task receive mail box size

Found in: [Component config](#) > [LWIP](#)

Set TCPIP task receive mail box size. Generally bigger value means higher throughput but more memory. The value should be bigger than UDP/TCP mail box size.

CONFIG_LWIP_DHCP_DOES_ARP_CHECK

DHCP: Perform ARP check on any offered address

Found in: [Component config](#) > [LWIP](#)

Enabling this option performs a check (via ARP request) if the offered IP address is not already in use by another host on the network.

CONFIG_LWIP_DHCP_DISABLE_CLIENT_ID

DHCP: Disable Use of HW address as client identification

Found in: [Component config](#) > [LWIP](#)

This option could be used to disable DHCP client identification with its MAC address. (Client id is used by DHCP servers to uniquely identify clients and are included in the DHCP packets as an option 61) Set this option to “y” in order to exclude option 61 from DHCP packets.

CONFIG_LWIP_DHCP_RESTORE_LAST_IP

DHCP: Restore last IP obtained from DHCP server

Found in: [Component config](#) > [LWIP](#)

When this option is enabled, DHCP client tries to re-obtain last valid IP address obtained from DHCP server. Last valid DHCP configuration is stored in nvs and restored after reset/power-up. If IP is still available, there is no need for sending discovery message to DHCP server and save some time.

DHCP server Contains:

- [CONFIG_LWIP_DHCPS_LEASE_UNIT](#)
- [CONFIG_LWIP_DHCPS_MAX_STATION_NUM](#)

CONFIG_LWIP_DHCPS_LEASE_UNIT

Multiplier for lease time, in seconds

Found in: [Component config](#) > [LWIP](#) > [DHCP server](#)

The DHCP server is calculating lease time multiplying the sent and received times by this number of seconds per unit. The default is 60, that equals one minute.

CONFIG_LWIP_DHCPS_MAX_STATION_NUM

Maximum number of stations

Found in: [Component config](#) > [LWIP](#) > [DHCP server](#)

The maximum number of DHCP clients that are connected to the server. After this number is exceeded, DHCP server removes of the oldest device from it's address pool, without notification.

CONFIG_LWIP_AUTOIP

Enable IPV4 Link-Local Addressing (AUTOIP)

Found in: [Component config](#) > [LWIP](#)

Enabling this option allows the device to self-assign an address in the 169.256/16 range if none is assigned statically or via DHCP.

See RFC 3927.

Contains:

- [CONFIG_LWIP_AUTOIP_TRIES](#)
- [CONFIG_LWIP_AUTOIP_MAX_CONFLICTS](#)
- [CONFIG_LWIP_AUTOIP_RATE_LIMIT_INTERVAL](#)

CONFIG_LWIP_AUTOIP_TRIES

DHCP Probes before self-assigning IPv4 LL address

Found in: [Component config](#) > [LWIP](#) > [CONFIG_LWIP_AUTOIP](#)

DHCP client will send this many probes before self-assigning a link local address.

From LWIP help: “This can be set as low as 1 to get an AutoIP address very quickly, but you should be prepared to handle a changing IP address when DHCP overrides AutoIP.” (In the case of ESP-IDF, this means multiple SYSTEM_EVENT_STA_GOT_IP events.)

CONFIG_LWIP_AUTOIP_MAX_CONFLICTS

Max IP conflicts before rate limiting

Found in: [Component config](#) > [LWIP](#) > [CONFIG_LWIP_AUTOIP](#)

If the AUTOIP functionality detects this many IP conflicts while self-assigning an address, it will go into a rate limited mode.

CONFIG_LWIP_AUTOIP_RATE_LIMIT_INTERVAL

Rate limited interval (seconds)

Found in: [Component config](#) > [LWIP](#) > [CONFIG_LWIP_AUTOIP](#)

If rate limiting self-assignment requests, wait this long between each request.

CONFIG_LWIP_IPV6_AUTOCONFIG

Enable IPV6 stateless address autoconfiguration

Found in: [Component config](#) > [LWIP](#)

Enabling this option allows the devices to IPV6 stateless address autoconfiguration.

See RFC 4862.

CONFIG_LWIP_NETIF_LOOPBACK

Support per-interface loopback

Found in: [Component config](#) > [LWIP](#)

Enabling this option means that if a packet is sent with a destination address equal to the interface's own IP address, it will “loop back” and be received by this interface.

Contains:

- [CONFIG_LWIP_LOOPBACK_MAX_PBUFS](#)

CONFIG_LWIP_LOOPBACK_MAX_PBUFS

Max queued loopback packets per interface

Found in: [Component config](#) > [LWIP](#) > [CONFIG_LWIP_NETIF_LOOPBACK](#)

Configure the maximum number of packets which can be queued for loopback on a given interface. Reducing this number may cause packets to be dropped, but will avoid filling memory with queued packet data.

TCP Contains:

- [CONFIG_LWIP_TCP_ISN_HOOK](#)
- [CONFIG_LWIP_MAX_ACTIVE_TCP](#)
- [CONFIG_LWIP_MAX_LISTENING_TCP](#)
- [CONFIG_LWIP_TCP_HIGH_SPEED_RETRANSMISSION](#)
- [CONFIG_LWIP_TCP_MAXRTX](#)
- [CONFIG_LWIP_TCP_SYNMAXRTX](#)
- [CONFIG_LWIP_TCP_MSS](#)
- [CONFIG_LWIP_TCP_TMR_INTERVAL](#)
- [CONFIG_LWIP_TCP_MSL](#)
- [CONFIG_LWIP_TCP_SND_BUF_DEFAULT](#)
- [CONFIG_LWIP_TCP_WND_DEFAULT](#)
- [CONFIG_LWIP_TCP_RECVMBOX_SIZE](#)
- [CONFIG_LWIP_TCP_QUEUE_OOSEQ](#)
- [CONFIG_LWIP_TCP_SACK_OUT](#)
- [CONFIG_LWIP_TCP_KEEP_CONNECTION_WHEN_IP_CHANGES](#)
- [CONFIG_LWIP_TCP_OVERSIZE](#)
- [CONFIG_LWIP_WND_SCALE](#)
- [CONFIG_LWIP_TCP_RTO_TIME](#)

CONFIG_LWIP_TCP_ISN_HOOK

Enable TCP ISN Hook

Found in: [Component config](#) > [LWIP](#) > [TCP](#)

Enables custom TCP ISN hook to randomize initial sequence number in TCP connection. This is recommended as default lwIP implementation (*tcp_next_iss*) is not very strong, as it does not take into consideration any platform specific entropy source.

CONFIG_LWIP_MAX_ACTIVE_TCP

Maximum active TCP Connections

Found in: [Component config](#) > [LWIP](#) > [TCP](#)

The maximum number of simultaneously active TCP connections. The practical maximum limit is determined by available heap memory at runtime.

Changing this value by itself does not substantially change the memory usage of LWIP, except for preventing new TCP connections after the limit is reached.

CONFIG_LWIP_MAX_LISTENING_TCP

Maximum listening TCP Connections

Found in: [Component config](#) > [LWIP](#) > [TCP](#)

The maximum number of simultaneously listening TCP connections. The practical maximum limit is determined by available heap memory at runtime.

Changing this value by itself does not substantially change the memory usage of LWIP, except for preventing new listening TCP connections after the limit is reached.

CONFIG_LWIP_TCP_HIGH_SPEED_RETRANSMISSION

TCP high speed retransmissions

Found in: [Component config](#) > [LWIP](#) > [TCP](#)

Speed up the TCP retransmission interval. If disabled, it is recommended to change the number of SYN retransmissions to 6, and TCP initial rto time to 3000.

CONFIG_LWIP_TCP_MAXRTX

Maximum number of retransmissions of data segments

Found in: [Component config](#) > [LWIP](#) > [TCP](#)

Set maximum number of retransmissions of data segments.

CONFIG_LWIP_TCP_SYNMAXRTX

Maximum number of retransmissions of SYN segments

Found in: [Component config](#) > [LWIP](#) > [TCP](#)

Set maximum number of retransmissions of SYN segments.

CONFIG_LWIP_TCP_MSS

Maximum Segment Size (MSS)

Found in: [Component config](#) > [LWIP](#) > [TCP](#)

Set maximum segment size for TCP transmission.

Can be set lower to save RAM, the default value 1460(ipv4)/1440(ipv6) will give best throughput. IPv4 TCP_MSS Range: 576 <= TCP_MSS <= 1460 IPv6 TCP_MSS Range: 1220 <= TCP_mSS <= 1440

CONFIG_LWIP_TCP_TMR_INTERVAL

TCP timer interval(ms)

Found in: [Component config](#) > [LWIP](#) > [TCP](#)

Set TCP timer interval in milliseconds.

Can be used to speed connections on bad networks. A lower value will redeliver unacked packets faster.

CONFIG_LWIP_TCP_MSL

Maximum segment lifetime (MSL)

Found in: [Component config](#) > [LWIP](#) > [TCP](#)

Set maximum segment lifetime in in milliseconds.

CONFIG_LWIP_TCP_SND_BUF_DEFAULT

Default send buffer size

Found in: [Component config](#) > [LWIP](#) > [TCP](#)

Set default send buffer size for new TCP sockets.

Per-socket send buffer size can be changed at runtime with `lwip_setsockopt(s, TCP_SNDBUF, ...)`.

This value must be at least 2x the MSS size, and the default is 4x the default MSS size.

Setting a smaller default SNDBUF size can save some RAM, but will decrease performance.

CONFIG_LWIP_TCP_WND_DEFAULT

Default receive window size

Found in: [Component config](#) > [LWIP](#) > [TCP](#)

Set default TCP receive window size for new TCP sockets.

Per-socket receive window size can be changed at runtime with `lwip_setsockopt(s, TCP_WINDOW, ...)`.

Setting a smaller default receive window size can save some RAM, but will significantly decrease performance.

CONFIG_LWIP_TCP_RECVMBOX_SIZE

Default TCP receive mail box size

Found in: [Component config](#) > [LWIP](#) > [TCP](#)

Set TCP receive mail box size. Generally bigger value means higher throughput but more memory. The recommended value is: $LWIP_TCP_WND_DEFAULT/TCP_MSS + 2$, e.g. if $LWIP_TCP_WND_DEFAULT=14360$, $TCP_MSS=1436$, then the recommended receive mail box size is $(14360/1436 + 2) = 12$.

TCP receive mail box is a per socket mail box, when the application receives packets from TCP socket, LWIP core firstly posts the packets to TCP receive mail box and the application then fetches the packets from mail box. It means LWIP can cache maximum $LWIP_TCP_RECCVMBOX_SIZE$ packets for each TCP socket, so the maximum possible cached TCP packets for all TCP sockets is $LWIP_TCP_RECCVMBOX_SIZE$ multiples the maximum TCP socket number. In other words, the bigger $LWIP_TCP_RECCVMBOX_SIZE$ means more memory. On the other hand, if the receive mail box is too small, the mail box may be full. If the mail box is full, the LWIP drops the packets. So generally we need to make sure the TCP receive mail box is big enough to avoid packet drop between LWIP core and application.

CONFIG_LWIP_TCP_QUEUE_OOSEQ

Queue incoming out-of-order segments

Found in: [Component config](#) > [LWIP](#) > [TCP](#)

Queue incoming out-of-order segments for later use.

Disable this option to save some RAM during TCP sessions, at the expense of increased retransmissions if segments arrive out of order.

CONFIG_LWIP_TCP_SACK_OUT

Support sending selective acknowledgements

Found in: [Component config](#) > [LWIP](#) > [TCP](#)

TCP will support sending selective acknowledgements (SACKs).

CONFIG_LWIP_TCP_KEEP_CONNECTION_WHEN_IP_CHANGES

Keep TCP connections when IP changed

Found in: [Component config](#) > [LWIP](#) > [TCP](#)

This option is enabled when the following scenario happen: network dropped and reconnected, IP changes is like: 192.168.0.2->0.0.0.0->192.168.0.2

Disable this option to keep consistent with the original LWIP code behavior.

CONFIG_LWIP_TCP_OVERSIZE

Pre-allocate transmit PBUF size

Found in: [Component config](#) > [LWIP](#) > [TCP](#)

Allows enabling “oversize” allocation of TCP transmission pbufs ahead of time, which can reduce the length of pbuf chains used for transmission.

This will not make a difference to sockets where Nagle’s algorithm is disabled.

Default value of MSS is fine for most applications, 25% MSS may save some RAM when only transmitting small amounts of data. Disabled will have worst performance and fragmentation characteristics, but uses least RAM overall.

Available options:

- MSS (LWIP_TCP_OVERSIZE_MSS)
- 25% MSS (LWIP_TCP_OVERSIZE_QUARTER_MSS)
- Disabled (LWIP_TCP_OVERSIZE_DISABLE)

CONFIG_LWIP_WND_SCALE

Support TCP window scale

Found in: [Component config](#) > [LWIP](#) > [TCP](#)

Enable this feature to support TCP window scaling.

CONFIG_LWIP_TCP_RCV_SCALE

Set TCP receiving window scaling factor

Found in: [Component config](#) > [LWIP](#) > [TCP](#) > [CONFIG_LWIP_WND_SCALE](#)

Enable this feature to support TCP window scaling.

CONFIG_LWIP_TCP_RTO_TIME

Default TCP rto time

Found in: [Component config](#) > [LWIP](#) > [TCP](#)

Set default TCP rto time for a reasonable initial rto. In bad network environment, recommend set value of rto time to 1500.

UDP Contains:

- [CONFIG_LWIP_MAX_UDP_PCBS](#)
- [CONFIG_LWIP_UDP_RECVMBOX_SIZE](#)

CONFIG_LWIP_MAX_UDP_PCBS

Maximum active UDP control blocks

Found in: [Component config](#) > [LWIP](#) > [UDP](#)

The maximum number of active UDP “connections” (ie UDP sockets sending/receiving data). The practical maximum limit is determined by available heap memory at runtime.

CONFIG_LWIP_UDP_RECVMBOX_SIZE

Default UDP receive mail box size

Found in: [Component config](#) > [LWIP](#) > [UDP](#)

Set UDP receive mail box size. The recommended value is 6.

UDP receive mail box is a per socket mail box, when the application receives packets from UDP socket, LWIP core firstly posts the packets to UDP receive mail box and the application then fetches the packets from mail box. It means LWIP can caches maximum `UDP_RECCVMBOX_SIZE` packets for each UDP socket, so the maximum possible cached UDP packets for all UDP sockets is `UDP_RECCVMBOX_SIZE` multiplies the maximum UDP socket number. In other words, the bigger `UDP_RECVMBOX_SIZE` means more memory. On the other hand, if the receive mail box is too small, the mail box may be full. If the mail box is full, the LWIP drops the packets. So generally we need to make sure the UDP receive mail box is big enough to avoid packet drop between LWIP core and application.

CONFIG_LWIP_TCPIP_TASK_STACK_SIZE

TCP/IP Task Stack Size

Found in: [Component config](#) > [LWIP](#)

Configure TCP/IP task stack size, used by LWIP to process multi-threaded TCP/IP operations. Setting this stack too small will result in stack overflow crashes.

CONFIG_LWIP_TCPIP_TASK_AFFINITY

TCP/IP task affinity

Found in: [Component config](#) > [LWIP](#)

Allows setting LwIP tasks affinity, i.e. whether the task is pinned to CPU0, pinned to CPU1, or allowed to run on any CPU. Currently this applies to “TCP/IP” task and “Ping” task.

Available options:

- No affinity (`LWIP_TCPIP_TASK_AFFINITY_NO_AFFINITY`)
- CPU0 (`LWIP_TCPIP_TASK_AFFINITY_CPU0`)
- CPU1 (`LWIP_TCPIP_TASK_AFFINITY_CPU1`)

CONFIG_LWIP_PPP_SUPPORT

Enable PPP support (new/experimental)

Found in: [Component config](#) > [LWIP](#)

Enable PPP stack. Now only PPP over serial is possible.

PPP over serial support is experimental and unsupported.

Contains:

- [CONFIG_LWIP_PPP_ENABLE_IPV6](#)

CONFIG_LWIP_PPP_ENABLE_IPV6

Enable IPV6 support for PPP connections (IPV6CP)

Found in: [Component config](#) > [LWIP](#) > [CONFIG_LWIP_PPP_SUPPORT](#)

Enable IPV6 support in PPP for the local link between the DTE (processor) and DCE (modem). There are some modems which do not support the IPV6 addressing in the local link. If they are requested for IPV6CP negotiation, they may time out. This would in turn fail the configuration for the whole link. If your modem is not responding correctly to PPP Phase Network, try to disable IPV6 support.

CONFIG_LWIP_IPV6_MEMP_NUM_ND6_QUEUE

Max number of IPv6 packets to queue during MAC resolution

Found in: [Component config](#) > [LWIP](#)

Config max number of IPv6 packets to queue during MAC resolution.

CONFIG_LWIP_IPV6_ND6_NUM_NEIGHBORS

Max number of entries in IPv6 neighbor cache

Found in: [Component config](#) > [LWIP](#)

Config max number of entries in IPv6 neighbor cache

CONFIG_LWIP_PPP_NOTIFY_PHASE_SUPPORT

Enable Notify Phase Callback

Found in: [Component config](#) > [LWIP](#)

Enable to set a callback which is called on change of the internal PPP state machine.

CONFIG_LWIP_PPP_PAP_SUPPORT

Enable PAP support

Found in: [Component config](#) > [LWIP](#)

Enable Password Authentication Protocol (PAP) support

CONFIG_LWIP_PPP_CHAP_SUPPORT

Enable CHAP support

Found in: [Component config](#) > [LWIP](#)

Enable Challenge Handshake Authentication Protocol (CHAP) support

CONFIG_LWIP_PPP_MSCHAP_SUPPORT

Enable MSCHAP support

Found in: [Component config](#) > [LWIP](#)

Enable Microsoft version of the Challenge-Handshake Authentication Protocol (MSCHAP) support

CONFIG_LWIP_PPP_MPPE_SUPPORT

Enable MPPE support

Found in: [Component config](#) > [LWIP](#)

Enable Microsoft Point-to-Point Encryption (MPPE) support

CONFIG_LWIP_PPP_DEBUG_ON

Enable PPP debug log output

Found in: [Component config](#) > [LWIP](#)

Enable PPP debug log output

ICMP Contains:

- [CONFIG_LWIP_MULTICAST_PING](#)
- [CONFIG_LWIP_BROADCAST_PING](#)

CONFIG_LWIP_MULTICAST_PING

Respond to multicast pings

Found in: [Component config](#) > [LWIP](#) > [ICMP](#)

CONFIG_LWIP_BROADCAST_PING

Respond to broadcast pings

Found in: [Component config](#) > [LWIP](#) > [ICMP](#)

LWIP RAW API Contains:

- [CONFIG_LWIP_MAX_RAW_PCBS](#)

CONFIG_LWIP_MAX_RAW_PCBS

Maximum LWIP RAW PCBs

Found in: [Component config](#) > [LWIP](#) > [LWIP RAW API](#)

The maximum number of simultaneously active LWIP RAW protocol control blocks. The practical maximum limit is determined by available heap memory at runtime.

SNTP Contains:

- [CONFIG_LWIP_DHCP_MAX_NTP_SERVERS](#)
- [CONFIG_LWIP_SNTP_UPDATE_DELAY](#)

CONFIG_LWIP_DHCP_MAX_NTP_SERVERS

Maximum number of NTP servers

Found in: [Component config](#) > [LWIP](#) > [SNTP](#)

Set maximum number of NTP servers used by LwIP SNTP module. First argument of `sntp_setserver/sntp_setservername` functions is limited to this value.

CONFIG_LWIP_SNTP_UPDATE_DELAY

Request interval to update time (ms)

Found in: *Component config* > *LWIP* > *SNTP*

This option allows you to set the time update period via SNTP. Default is 1 hour. Must not be below 15 seconds by specification. (SNTPv4 RFC 4330 enforces a minimum update time of 15 seconds).

mbedTLS Contains:

- *CONFIG_MBEDTLS_MEM_ALLOC_MODE*
- *CONFIG_MBEDTLS_SSL_MAX_CONTENT_LEN*
- *CONFIG_MBEDTLS_ASYMMETRIC_CONTENT_LEN*
- *CONFIG_MBEDTLS_DEBUG*
- *CONFIG_MBEDTLS_ECP_RESTARTABLE*
- *CONFIG_MBEDTLS_CMAC_C*
- *CONFIG_MBEDTLS_HARDWARE_AES*
- *CONFIG_MBEDTLS_HARDWARE_MPI*
- *CONFIG_MBEDTLS_HARDWARE_SHA*
- *CONFIG_MBEDTLS_HAVE_TIME*
- *CONFIG_MBEDTLS_TLS_MODE*
- *TLS Key Exchange Methods*
- *CONFIG_MBEDTLS_SSL_RENEGOTIATION*
- *CONFIG_MBEDTLS_SSL_PROTO_SSL3*
- *CONFIG_MBEDTLS_SSL_PROTO_TLS1*
- *CONFIG_MBEDTLS_SSL_PROTO_TLS1_1*
- *CONFIG_MBEDTLS_SSL_PROTO_TLS1_2*
- *CONFIG_MBEDTLS_SSL_PROTO_DTLS*
- *CONFIG_MBEDTLS_SSL_ALPN*
- *CONFIG_MBEDTLS_CLIENT_SSL_SESSION_TICKETS*
- *CONFIG_MBEDTLS_SERVER_SSL_SESSION_TICKETS*
- *Symmetric Ciphers*
- *CONFIG_MBEDTLS_RIPEMD160_C*
- *Certificates*
- *CONFIG_MBEDTLS_ECP_C*
- *CONFIG_MBEDTLS_SECURITY_RISKS*

CONFIG_MBEDTLS_MEM_ALLOC_MODE

Memory allocation strategy

Found in: *Component config* > *mbedTLS*

Allocation strategy for mbedTLS, essentially provides ability to allocate all required dynamic allocations from,

- Internal DRAM memory only
- External SPIRAM memory only
- Either internal or external memory based on default malloc() behavior in ESP-IDF
- Custom allocation mode, by overwriting calloc()/free() using mbedtls_platform_set_malloc_free() function

Recommended mode here is always internal, since that is most preferred from security perspective. But if application requirement does not allow sufficient free internal memory then alternate mode can be selected.

Available options:

- Internal memory (MBEDTLS_INTERNAL_MEM_ALLOC)
- External SPIRAM (MBEDTLS_EXTERNAL_MEM_ALLOC)
- Default alloc mode (MBEDTLS_DEFAULT_MEM_ALLOC)
- Custom alloc mode (MBEDTLS_CUSTOM_MEM_ALLOC)

CONFIG_MBEDTLS_SSL_MAX_CONTENT_LEN

TLS maximum message content length

Found in: [Component config](#) > [mbedtls](#)

Maximum TLS message length (in bytes) supported by mbedtls.

16384 is the default and this value is required to comply fully with TLS standards.

However you can set a lower value in order to save RAM. This is safe if the other end of the connection supports Maximum Fragment Length Negotiation Extension (`max_fragment_length`, see RFC6066) or you know for certain that it will never send a message longer than a certain number of bytes.

If the value is set too low, symptoms are a failed TLS handshake or a return value of `MBEDTLS_ERR_SSL_INVALID_RECORD` (-0x7200).

CONFIG_MBEDTLS_ASYMMETRIC_CONTENT_LEN

Asymmetric in/out fragment length

Found in: [Component config](#) > [mbedtls](#)

If enabled, this option allows customizing TLS in/out fragment length in asymmetric way. Please note that enabling this with default values saves 12KB of dynamic memory per TLS connection.

CONFIG_MBEDTLS_SSL_IN_CONTENT_LEN

TLS maximum incoming fragment length

Found in: [Component config](#) > [mbedtls](#) > [CONFIG_MBEDTLS_ASYMMETRIC_CONTENT_LEN](#)

This defines maximum incoming fragment length, overriding default maximum content length (`MBEDTLS_SSL_MAX_CONTENT_LEN`).

CONFIG_MBEDTLS_SSL_OUT_CONTENT_LEN

TLS maximum outgoing fragment length

Found in: [Component config](#) > [mbedtls](#) > [CONFIG_MBEDTLS_ASYMMETRIC_CONTENT_LEN](#)

This defines maximum outgoing fragment length, overriding default maximum content length (`MBEDTLS_SSL_MAX_CONTENT_LEN`).

CONFIG_MBEDTLS_DEBUG

Enable mbedtls debugging

Found in: [Component config](#) > [mbedtls](#)

Enable mbedtls debugging functions at compile time.

If this option is enabled, you can include “`mbedtls/esp_debug.h`” and call `mbedtls_esp_enable_debug_log()` at runtime in order to enable mbedtls debug output via the ESP log mechanism.

CONFIG_MBEDTLS_DEBUG_LEVEL

Set mbedtls debugging level

Found in: [Component config](#) > [mbedtls](#) > [CONFIG_MBEDTLS_DEBUG](#)

Set mbedtls debugging level

Available options:

- Warning (`MBEDTLS_DEBUG_LEVEL_WARN`)

- Info (MBEDTLS_DEBUG_LEVEL_INFO)
- Debug (MBEDTLS_DEBUG_LEVEL_DEBUG)
- Verbose (MBEDTLS_DEBUG_LEVEL_VERBOSE)

CONFIG_MBEDTLS_ECP_RESTARTABLE

Enable mbedTLS ecp restartable

Found in: [Component config](#) > [mbedtls](#)

Enable “non-blocking” ECC operations that can return early and be resumed.

CONFIG_MBEDTLS_CMAC_C

Enable CMAC mode for block ciphers

Found in: [Component config](#) > [mbedtls](#)

Enable the CMAC (Cipher-based Message Authentication Code) mode for block ciphers.

CONFIG_MBEDTLS_HARDWARE_AES

Enable hardware AES acceleration

Found in: [Component config](#) > [mbedtls](#)

Enable hardware accelerated AES encryption & decryption.

Note that if the ESP32 CPU is running at 240MHz, hardware AES does not offer any speed boost over software AES.

CONFIG_MBEDTLS_HARDWARE_MPI

Enable hardware MPI (bignum) acceleration

Found in: [Component config](#) > [mbedtls](#)

Enable hardware accelerated multiple precision integer operations.

Hardware accelerated multiplication, modulo multiplication, and modular exponentiation for up to 4096 bit results.

These operations are used by RSA.

CONFIG_MBEDTLS_HARDWARE_SHA

Enable hardware SHA acceleration

Found in: [Component config](#) > [mbedtls](#)

Enable hardware accelerated SHA1, SHA256, SHA384 & SHA512 in mbedTLS.

Due to a hardware limitation, hardware acceleration is only guaranteed if SHA digests are calculated one at a time. If more than one SHA digest is calculated at the same time, one will be calculated fully in hardware and the rest will be calculated (at least partially calculated) in software. This happens automatically.

SHA hardware acceleration is faster than software in some situations but slower in others. You should benchmark to find the best setting for you.

CONFIG_MBEDTLS_HAVE_TIME

Enable mbedtls time

Found in: [Component config](#) > [mbedtls](#)

System has time.h and time(). The time does not need to be correct, only time differences are used.

CONFIG_MBEDTLS_HAVE_TIME_DATE

Enable mbedtls certificate expiry check

Found in: [Component config](#) > [mbedtls](#) > [CONFIG_MBEDTLS_HAVE_TIME](#)

System has time.h and time(), gmtime() and the clock is correct. The time needs to be correct (not necessarily very accurate, but at least the date should be correct). This is used to verify the validity period of X.509 certificates.

It is suggested that you should get the real time by “SNTP” .

CONFIG_MBEDTLS_TLS_MODE

TLS Protocol Role

Found in: [Component config](#) > [mbedtls](#)

mbedtls can be compiled with protocol support for the TLS server, TLS client, or both server and client.

Reducing the number of TLS roles supported saves code size.

Available options:

- Server & Client (MBEDTLS_TLS_SERVER_AND_CLIENT)
- Server (MBEDTLS_TLS_SERVER_ONLY)
- Client (MBEDTLS_TLS_CLIENT_ONLY)
- None (MBEDTLS_TLS_DISABLED)

TLS Key Exchange Methods

 Contains:

- [CONFIG_MBEDTLS_PSK_MODES](#)
- [CONFIG_MBEDTLS_KEY_EXCHANGE_RSA](#)
- [CONFIG_MBEDTLS_KEY_EXCHANGE_DHE_RSA](#)
- [CONFIG_MBEDTLS_KEY_EXCHANGE_ELLIPTIC_CURVE](#)

CONFIG_MBEDTLS_PSK_MODES

Enable pre-shared-key ciphersuites

Found in: [Component config](#) > [mbedtls](#) > [TLS Key Exchange Methods](#)

Enable to show configuration for different types of pre-shared-key TLS authentication methods.

Leaving this options disabled will save code size if they are not used.

CONFIG_MBEDTLS_KEY_EXCHANGE_PSK

Enable PSK based ciphersuite modes

Found in: [Component config](#) > [mbedtls](#) > [TLS Key Exchange Methods](#) > [CONFIG_MBEDTLS_PSK_MODES](#)

Enable to support symmetric key PSK (pre-shared-key) TLS key exchange modes.

CONFIG_MBEDTLS_KEY_EXCHANGE_DHE_PSK

Enable DHE-PSK based ciphersuite modes

Found in: [Component config](#) > [mbedtls](#) > [TLS Key Exchange Methods](#) > [CONFIG_MBEDTLS_PSK_MODES](#)

Enable to support Diffie-Hellman PSK (pre-shared-key) TLS authentication modes.

CONFIG_MBEDTLS_KEY_EXCHANGE_ECDHE_PSK

Enable ECDHE-PSK based ciphersuite modes

Found in: [Component config](#) > [mbedtls](#) > [TLS Key Exchange Methods](#) > [CONFIG_MBEDTLS_PSK_MODES](#)

Enable to support Elliptic-Curve-Diffie-Hellman PSK (pre-shared-key) TLS authentication modes.

CONFIG_MBEDTLS_KEY_EXCHANGE_RSA_PSK

Enable RSA-PSK based ciphersuite modes

Found in: [Component config](#) > [mbedtls](#) > [TLS Key Exchange Methods](#) > [CONFIG_MBEDTLS_PSK_MODES](#)

Enable to support RSA PSK (pre-shared-key) TLS authentication modes.

CONFIG_MBEDTLS_KEY_EXCHANGE_RSA

Enable RSA-only based ciphersuite modes

Found in: [Component config](#) > [mbedtls](#) > [TLS Key Exchange Methods](#)

Enable to support ciphersuites with prefix TLS-RSA-WITH-

CONFIG_MBEDTLS_KEY_EXCHANGE_DHE_RSA

Enable DHE-RSA based ciphersuite modes

Found in: [Component config](#) > [mbedtls](#) > [TLS Key Exchange Methods](#)

Enable to support ciphersuites with prefix TLS-DHE-RSA-WITH-

CONFIG_MBEDTLS_KEY_EXCHANGE_ELLIPTIC_CURVE

Support Elliptic Curve based ciphersuites

Found in: [Component config](#) > [mbedtls](#) > [TLS Key Exchange Methods](#)

Enable to show Elliptic Curve based ciphersuite mode options.

Disabling all Elliptic Curve ciphersuites saves code size and can give slightly faster TLS handshakes, provided the server supports RSA-only ciphersuite modes.

CONFIG_MBEDTLS_KEY_EXCHANGE_ECDHE_RSA

Enable ECDHE-RSA based ciphersuite modes

Found in: [Component config](#) > [mbedtls](#) > [TLS Key Exchange Methods](#) > [CONFIG_MBEDTLS_KEY_EXCHANGE_ELLIPTIC_CURVE](#)

Enable to support ciphersuites with prefix TLS-ECDHE-RSA-WITH-

CONFIG_MBEDTLS_KEY_EXCHANGE_ECDHE_ECDSA

Enable ECDHE-ECDSA based ciphersuite modes

Found in: [Component config](#) > [mbedtls](#) > [TLS Key Exchange Methods](#) > [CONFIG_MBEDTLS_KEY_EXCHANGE_ELLIPTIC_CURVE](#)

Enable to support ciphersuites with prefix TLS-ECDHE-RSA-WITH-

CONFIG_MBEDTLS_KEY_EXCHANGE_ECDH_ECDSA

Enable ECDH-ECDSA based ciphersuite modes

Found in: [Component config](#) > [mbedtls](#) > [TLS Key Exchange Methods](#) > [CONFIG_MBEDTLS_KEY_EXCHANGE_ELLIPTIC_CURVE](#)

Enable to support ciphersuites with prefix TLS-ECDHE-RSA-WITH-

CONFIG_MBEDTLS_KEY_EXCHANGE_ECDH_RSA

Enable ECDH-RSA based ciphersuite modes

Found in: [Component config](#) > [mbedtls](#) > [TLS Key Exchange Methods](#) > [CONFIG_MBEDTLS_KEY_EXCHANGE_ELLIPTIC_CURVE](#)

Enable to support ciphersuites with prefix TLS-ECDHE-RSA-WITH-

CONFIG_MBEDTLS_SSL_RENEGOTIATION

Support TLS renegotiation

Found in: [Component config](#) > [mbedtls](#)

The two main uses of renegotiation are (1) refresh keys on long-lived connections and (2) client authentication after the initial handshake. If you don't need renegotiation, disabling it will save code size and reduce the possibility of abuse/vulnerability.

CONFIG_MBEDTLS_SSL_PROTO_SSL3

Legacy SSL 3.0 support

Found in: [Component config](#) > [mbedtls](#)

Support the legacy SSL 3.0 protocol. Most servers will speak a newer TLS protocol these days.

CONFIG_MBEDTLS_SSL_PROTO_TLS1

Support TLS 1.0 protocol

Found in: [Component config](#) > [mbedtls](#)

CONFIG_MBEDTLS_SSL_PROTO_TLS1_1

Support TLS 1.1 protocol

Found in: [Component config](#) > [mbedtls](#)

CONFIG_MBEDTLS_SSL_PROTO_TLS1_2

Support TLS 1.2 protocol

Found in: [Component config](#) > [mbedtls](#)

CONFIG_MBEDTLS_SSL_PROTO_DTLS

Support DTLS protocol (all versions)

Found in: [Component config](#) > [mbedtls](#)

Requires TLS 1.1 to be enabled for DTLS 1.0 Requires TLS 1.2 to be enabled for DTLS 1.2

CONFIG_MBEDTLS_SSL_ALPN

Support ALPN (Application Layer Protocol Negotiation)

Found in: [Component config](#) > [mbedtls](#)

Disabling this option will save some code size if it is not needed.

CONFIG_MBEDTLS_CLIENT_SSL_SESSION_TICKETS

TLS: Client Support for RFC 5077 SSL session tickets

Found in: [Component config](#) > [mbedtls](#)

Client support for RFC 5077 session tickets. See mbedtls documentation for more details. Disabling this option will save some code size.

CONFIG_MBEDTLS_SERVER_SSL_SESSION_TICKETS

TLS: Server Support for RFC 5077 SSL session tickets

Found in: [Component config](#) > [mbedtls](#)

Server support for RFC 5077 session tickets. See mbedtls documentation for more details. Disabling this option will save some code size.

Symmetric Ciphers Contains:

- [CONFIG_MBEDTLS_AES_C](#)
- [CONFIG_MBEDTLS_CAMELLIA_C](#)
- [CONFIG_MBEDTLS_DES_C](#)
- [CONFIG_MBEDTLS_RC4_MODE](#)
- [CONFIG_MBEDTLS_BLOWFISH_C](#)
- [CONFIG_MBEDTLS_XTEA_C](#)
- [CONFIG_MBEDTLS_CCM_C](#)
- [CONFIG_MBEDTLS_GCM_C](#)

CONFIG_MBEDTLS_AES_C

AES block cipher

Found in: [Component config](#) > [mbedtls](#) > [Symmetric Ciphers](#)

CONFIG_MBEDTLS_CAMELLIA_C

Camellia block cipher

Found in: [Component config](#) > [mbedtls](#) > [Symmetric Ciphers](#)

CONFIG_MBEDTLS_DES_C

DES block cipher (legacy, insecure)

Found in: [Component config](#) > [mbedtls](#) > [Symmetric Ciphers](#)

Enables the DES block cipher to support 3DES-based TLS ciphersuites.

3DES is vulnerable to the Sweet32 attack and should only be enabled if absolutely necessary.

CONFIG_MBEDTLS_RC4_MODE

RC4 Stream Cipher (legacy, insecure)

Found in: [Component config](#) > [mbedtls](#) > [Symmetric Ciphers](#)

ARCFOUR (RC4) stream cipher can be disabled entirely, enabled but not added to default ciphersuites, or enabled completely.

Please consider the security implications before enabling RC4.

Available options:

- Disabled (MBEDTLS_RC4_DISABLED)
- Enabled, not in default ciphersuites (MBEDTLS_RC4_ENABLED_NO_DEFAULT)
- Enabled (MBEDTLS_RC4_ENABLED)

CONFIG_MBEDTLS_BLOWFISH_C

Blowfish block cipher (read help)

Found in: [Component config](#) > [mbedtls](#) > [Symmetric Ciphers](#)

Enables the Blowfish block cipher (not used for TLS sessions.)

The Blowfish cipher is not used for mbedtls TLS sessions but can be used for other purposes. Read up on the limitations of Blowfish (including Sweet32) before enabling.

CONFIG_MBEDTLS_XTEA_C

XTEA block cipher

Found in: [Component config](#) > [mbedtls](#) > [Symmetric Ciphers](#)

Enables the XTEA block cipher.

CONFIG_MBEDTLS_CCM_C

CCM (Counter with CBC-MAC) block cipher modes

Found in: [Component config](#) > [mbedtls](#) > [Symmetric Ciphers](#)

Enable Counter with CBC-MAC (CCM) modes for AES and/or Camellia ciphers.

Disabling this option saves some code size.

CONFIG_MBEDTLS_GCM_C

GCM (Galois/Counter) block cipher modes

Found in: [Component config](#) > [mbedtls](#) > [Symmetric Ciphers](#)

Enable Galois/Counter Mode for AES and/or Camellia ciphers.

This option is generally faster than CCM.

CONFIG_MBEDTLS_RIPEMD160_C

Enable RIPEMD-160 hash algorithm

Found in: [Component config](#) > [mbedtls](#)

Enable the RIPEMD-160 hash algorithm.

Certificates Contains:

- [CONFIG_MBEDTLS_PEM_PARSE_C](#)
- [CONFIG_MBEDTLS_PEM_WRITE_C](#)
- [CONFIG_MBEDTLS_X509_CRL_PARSE_C](#)
- [CONFIG_MBEDTLS_X509_CSR_PARSE_C](#)

CONFIG_MBEDTLS_PEM_PARSE_C

Read & Parse PEM formatted certificates

Found in: [Component config](#) > [mbedtls](#) > [Certificates](#)

Enable decoding/parsing of PEM formatted certificates.

If your certificates are all in the simpler DER format, disabling this option will save some code size.

CONFIG_MBEDTLS_PEM_WRITE_C

Write PEM formatted certificates

Found in: [Component config](#) > [mbedtls](#) > [Certificates](#)

Enable writing of PEM formatted certificates.

If writing certificate data only in DER format, disabling this option will save some code size.

CONFIG_MBEDTLS_X509_CRL_PARSE_C

X.509 CRL parsing

Found in: [Component config](#) > [mbedtls](#) > [Certificates](#)

Support for parsing X.509 Certificate Revocation Lists.

CONFIG_MBEDTLS_X509_CSR_PARSE_C

X.509 CSR parsing

Found in: [Component config](#) > [mbedtls](#) > [Certificates](#)

Support for parsing X.509 Certificate Signing Requests

CONFIG_MBEDTLS_ECP_C

Elliptic Curve Ciphers

Found in: [Component config](#) > [mbedtls](#)

Contains:

- [CONFIG_MBEDTLS_ECDH_C](#)
- [CONFIG_MBEDTLS_ECP_DP_SECP192R1_ENABLED](#)
- [CONFIG_MBEDTLS_ECP_DP_SECP224R1_ENABLED](#)
- [CONFIG_MBEDTLS_ECP_DP_SECP256R1_ENABLED](#)
- [CONFIG_MBEDTLS_ECP_DP_SECP384R1_ENABLED](#)
- [CONFIG_MBEDTLS_ECP_DP_SECP521R1_ENABLED](#)

- [CONFIG_MBEDTLS_ECP_DP_SECP192K1_ENABLED](#)
- [CONFIG_MBEDTLS_ECP_DP_SECP224K1_ENABLED](#)
- [CONFIG_MBEDTLS_ECP_DP_SECP256K1_ENABLED](#)
- [CONFIG_MBEDTLS_ECP_DP_BP256R1_ENABLED](#)
- [CONFIG_MBEDTLS_ECP_DP_BP384R1_ENABLED](#)
- [CONFIG_MBEDTLS_ECP_DP_BP512R1_ENABLED](#)
- [CONFIG_MBEDTLS_ECP_DP_CURVE25519_ENABLED](#)
- [CONFIG_MBEDTLS_ECP_NIST_OPTIM](#)

CONFIG_MBEDTLS_ECDH_C

Elliptic Curve Diffie-Hellman (ECDH)

Found in: [Component config](#) > [mbedtls](#) > [CONFIG_MBEDTLS_ECP_C](#)

Enable ECDH. Needed to use ECDHE-xxx TLS ciphersuites.

CONFIG_MBEDTLS_ECDSA_C

Elliptic Curve DSA

Found in: [Component config](#) > [mbedtls](#) > [CONFIG_MBEDTLS_ECP_C](#) > [CONFIG_MBEDTLS_ECDH_C](#)

Enable ECDSA. Needed to use ECDSA-xxx TLS ciphersuites.

CONFIG_MBEDTLS_ECP_DP_SECP192R1_ENABLED

Enable SECP192R1 curve

Found in: [Component config](#) > [mbedtls](#) > [CONFIG_MBEDTLS_ECP_C](#)

Enable support for SECP192R1 Elliptic Curve.

CONFIG_MBEDTLS_ECP_DP_SECP224R1_ENABLED

Enable SECP224R1 curve

Found in: [Component config](#) > [mbedtls](#) > [CONFIG_MBEDTLS_ECP_C](#)

Enable support for SECP224R1 Elliptic Curve.

CONFIG_MBEDTLS_ECP_DP_SECP256R1_ENABLED

Enable SECP256R1 curve

Found in: [Component config](#) > [mbedtls](#) > [CONFIG_MBEDTLS_ECP_C](#)

Enable support for SECP256R1 Elliptic Curve.

CONFIG_MBEDTLS_ECP_DP_SECP384R1_ENABLED

Enable SECP384R1 curve

Found in: [Component config](#) > [mbedtls](#) > [CONFIG_MBEDTLS_ECP_C](#)

Enable support for SECP384R1 Elliptic Curve.

CONFIG_MBEDTLS_ECP_DP_SECP521R1_ENABLED

Enable SECP521R1 curve

Found in: [Component config](#) > [mbedtls](#) > [CONFIG_MBEDTLS_ECP_C](#)

Enable support for SECP521R1 Elliptic Curve.

CONFIG_MBEDTLS_ECP_DP_SECP192K1_ENABLED

Enable SECP192K1 curve

Found in: [Component config](#) > [mbedtls](#) > [CONFIG_MBEDTLS_ECP_C](#)

Enable support for SECP192K1 Elliptic Curve.

CONFIG_MBEDTLS_ECP_DP_SECP224K1_ENABLED

Enable SECP224K1 curve

Found in: [Component config](#) > [mbedtls](#) > [CONFIG_MBEDTLS_ECP_C](#)

Enable support for SECP224K1 Elliptic Curve.

CONFIG_MBEDTLS_ECP_DP_SECP256K1_ENABLED

Enable SECP256K1 curve

Found in: [Component config](#) > [mbedtls](#) > [CONFIG_MBEDTLS_ECP_C](#)

Enable support for SECP256K1 Elliptic Curve.

CONFIG_MBEDTLS_ECP_DP_BP256R1_ENABLED

Enable BP256R1 curve

Found in: [Component config](#) > [mbedtls](#) > [CONFIG_MBEDTLS_ECP_C](#)

support for DP Elliptic Curve.

CONFIG_MBEDTLS_ECP_DP_BP384R1_ENABLED

Enable BP384R1 curve

Found in: [Component config](#) > [mbedtls](#) > [CONFIG_MBEDTLS_ECP_C](#)

support for DP Elliptic Curve.

CONFIG_MBEDTLS_ECP_DP_BP512R1_ENABLED

Enable BP512R1 curve

Found in: [Component config](#) > [mbedtls](#) > [CONFIG_MBEDTLS_ECP_C](#)

support for DP Elliptic Curve.

CONFIG_MBEDTLS_ECP_DP_CURVE25519_ENABLED

Enable CURVE25519 curve

Found in: [Component config](#) > [mbedtls](#) > [CONFIG_MBEDTLS_ECP_C](#)

Enable support for CURVE25519 Elliptic Curve.

CONFIG_MBEDTLS_ECP_NIST_OPTIM

NIST ‘modulo p’ optimisations

Found in: [Component config](#) > [mbedtls](#) > [CONFIG_MBEDTLS_ECP_C](#)

NIST ‘modulo p’ optimisations increase Elliptic Curve operation performance.

Disabling this option saves some code size.

end of Elliptic Curve options

CONFIG_MBEDTLS_SECURITY_RISKS

Show configurations with potential security risks

Found in: [Component config](#) > [mbedtls](#)

Contains:

- [CONFIG_MBEDTLS_ALLOW_UNSUPPORTED_CRITICAL_EXT](#)

CONFIG_MBEDTLS_ALLOW_UNSUPPORTED_CRITICAL_EXT

X.509 CRT parsing with unsupported critical extensions

Found in: [Component config](#) > [mbedtls](#) > [CONFIG_MBEDTLS_SECURITY_RISKS](#)

Allow the X.509 certificate parser to load certificates with unsupported critical extensions

mDNS Contains:

- [CONFIG_MDNS_MAX_SERVICES](#)
- [CONFIG_MDNS_TASK_PRIORITY](#)
- [CONFIG_MDNS_TASK_AFFINITY](#)
- [CONFIG_MDNS_SERVICE_ADD_TIMEOUT_MS](#)
- [CONFIG_MDNS_TIMER_PERIOD_MS](#)

CONFIG_MDNS_MAX_SERVICES

Max number of services

Found in: [Component config](#) > [mDNS](#)

Services take up a certain amount of memory, and allowing fewer services to be open at the same time conserves memory. Specify the maximum amount of services here. The valid value is from 1 to 64.

CONFIG_MDNS_TASK_PRIORITY

mDNS task priority

Found in: [Component config](#) > [mDNS](#)

Allows setting mDNS task priority. Please do not set the task priority higher than priorities of system tasks. Compile time warning/error would be emitted if the chosen task priority were too high.

CONFIG_MDNS_TASK_AFFINITY

mDNS task affinity

Found in: [Component config](#) > [mDNS](#)

Allows setting mDNS tasks affinity, i.e. whether the task is pinned to CPU0, pinned to CPU1, or allowed to run on any CPU.

Available options:

- No affinity (MDNS_TASK_AFFINITY_NO_AFFINITY)
- CPU0 (MDNS_TASK_AFFINITY_CPU0)
- CPU1 (MDNS_TASK_AFFINITY_CPU1)

CONFIG_MDNS_SERVICE_ADD_TIMEOUT_MS

mDNS adding service timeout (ms)

Found in: [Component config](#) > [mDNS](#)

Configures timeout for adding a new mDNS service. Adding a service fails if could not be completed within this time.

CONFIG_MDNS_TIMER_PERIOD_MS

mDNS timer period (ms)

Found in: [Component config](#) > [mDNS](#)

Configures period of mDNS timer, which periodically transmits packets and schedules mDNS searches.

ESP-MQTT Configurations Contains:

- [CONFIG_MQTT_PROTOCOL_311](#)
- [CONFIG_MQTT_TRANSPORT_SSL](#)
- [CONFIG_MQTT_TRANSPORT_WEBSOCKET](#)
- [CONFIG_MQTT_USE_CUSTOM_CONFIG](#)
- [CONFIG_MQTT_TASK_CORE_SELECTION_ENABLED](#)
- [CONFIG_MQTT_CUSTOM_OUTBOX](#)

CONFIG_MQTT_PROTOCOL_311

Enable MQTT protocol 3.1.1

Found in: [Component config](#) > [ESP-MQTT Configurations](#)

If not, this library will use MQTT protocol 3.1

CONFIG_MQTT_TRANSPORT_SSL

Enable MQTT over SSL

Found in: [Component config](#) > [ESP-MQTT Configurations](#)

Enable MQTT transport over SSL with mbedtls

CONFIG_MQTT_TRANSPORT_WEBSOCKET

Enable MQTT over Websocket

Found in: [Component config](#) > [ESP-MQTT Configurations](#)

Enable MQTT transport over Websocket.

CONFIG_MQTT_TRANSPORT_WEBSOCKET_SECURE

Enable MQTT over Websocket Secure

Found in: [Component config](#) > [ESP-MQTT Configurations](#) > [CONFIG_MQTT_TRANSPORT_WEBSOCKET](#)

Enable MQTT transport over Websocket Secure.

CONFIG_MQTT_USE_CUSTOM_CONFIG

MQTT Using custom configurations

Found in: [Component config](#) > [ESP-MQTT Configurations](#)

Custom MQTT configurations.

CONFIG_MQTT_TCP_DEFAULT_PORT

Default MQTT over TCP port

Found in: [Component config](#) > [ESP-MQTT Configurations](#) > [CONFIG_MQTT_USE_CUSTOM_CONFIG](#)

Default MQTT over TCP port

CONFIG_MQTT_SSL_DEFAULT_PORT

Default MQTT over SSL port

Found in: [Component config](#) > [ESP-MQTT Configurations](#) > [CONFIG_MQTT_USE_CUSTOM_CONFIG](#)

Default MQTT over SSL port

CONFIG_MQTT_WS_DEFAULT_PORT

Default MQTT over Websocket port

Found in: [Component config](#) > [ESP-MQTT Configurations](#) > [CONFIG_MQTT_USE_CUSTOM_CONFIG](#)

Default MQTT over Websocket port

CONFIG_MQTT_WSS_DEFAULT_PORT

Default MQTT over Websocket Secure port

Found in: [Component config](#) > [ESP-MQTT Configurations](#) > [CONFIG_MQTT_USE_CUSTOM_CONFIG](#)

Default MQTT over Websocket Secure port

CONFIG_MQTT_BUFFER_SIZE

Default MQTT Buffer Size

Found in: [Component config](#) > [ESP-MQTT Configurations](#) > [CONFIG_MQTT_USE_CUSTOM_CONFIG](#)

This buffer size using for both transmit and receive

CONFIG_MQTT_TASK_STACK_SIZE

MQTT task stack size

Found in: [Component config](#) > [ESP-MQTT Configurations](#) > [CONFIG_MQTT_USE_CUSTOM_CONFIG](#)

MQTT task stack size

CONFIG_MQTT_DISABLE_API_LOCKS

Disable API locks

Found in: [Component config](#) > [ESP-MQTT Configurations](#) > [CONFIG_MQTT_USE_CUSTOM_CONFIG](#)

Default config employs API locks to protect internal structures. It is possible to disable these locks if the user code doesn't access MQTT API from multiple concurrent tasks

CONFIG_MQTT_TASK_PRIORITY

MQTT task priority

Found in: [Component config](#) > [ESP-MQTT Configurations](#) > [CONFIG_MQTT_USE_CUSTOM_CONFIG](#)

MQTT task priority. Higher number denotes higher priority.

CONFIG_MQTT_TASK_CORE_SELECTION_ENABLED

Enable MQTT task core selection

Found in: [Component config](#) > [ESP-MQTT Configurations](#)

This will enable core selection

CONFIG_MQTT_TASK_CORE_SELECTION

Core to use ?

Found in: [Component config](#) > [ESP-MQTT Configurations](#) > [CONFIG_MQTT_TASK_CORE_SELECTION_ENABLED](#)

Available options:

- Core 0 (MQTT_USE_CORE_0)
- Core 1 (MQTT_USE_CORE_1)

CONFIG_MQTT_CUSTOM_OUTBOX

Enable custom outbox implementation

Found in: [Component config](#) > [ESP-MQTT Configurations](#)

Set to true if a specific implementation of message outbox is needed (e.g. persistent outbox in NVM or similar).

Newlib Contains:

- [CONFIG_NEWLIB_STDOUT_LINE_ENDING](#)
- [CONFIG_NEWLIB_STDIN_LINE_ENDING](#)
- [CONFIG_NEWLIB_NANO_FORMAT](#)

CONFIG_NEWLIB_STDOUT_LINE_ENDING

Line ending for UART output

Found in: [Component config](#) > [Newlib](#)

This option allows configuring the desired line endings sent to UART when a newline ('n' , LF) appears on stdout. Three options are possible:

CRLF: whenever LF is encountered, prepend it with CR

LF: no modification is applied, stdout is sent as is

CR: each occurrence of LF is replaced with CR

This option doesn't affect behavior of the UART driver (drivers/uart.h).

Available options:

- CRLF (NEWLIB_STDOUT_LINE_ENDING_CRLF)
- LF (NEWLIB_STDOUT_LINE_ENDING_LF)
- CR (NEWLIB_STDOUT_LINE_ENDING_CR)

CONFIG_NEWLIB_STDIN_LINE_ENDING

Line ending for UART input

Found in: [Component config](#) > [Newlib](#)

This option allows configuring which input sequence on UART produces a newline (‘n’ , LF) on stdin. Three options are possible:

CRLF: CRLF is converted to LF

LF: no modification is applied, input is sent to stdin as is

CR: each occurrence of CR is replaced with LF

This option doesn’ t affect behavior of the UART driver (drivers/uart.h).

Available options:

- CRLF (NEWLIB_STDIN_LINE_ENDING_CRLF)
- LF (NEWLIB_STDIN_LINE_ENDING_LF)
- CR (NEWLIB_STDIN_LINE_ENDING_CR)

CONFIG_NEWLIB_NANO_FORMAT

Enable ‘nano’ formatting options for printf/scanf family

Found in: [Component config](#) > [Newlib](#)

ESP32 ROM contains parts of newlib C library, including printf/scanf family of functions. These functions have been compiled with so-called “nano” formatting option. This option doesn’ t support 64-bit integer formats and C99 features, such as positional arguments.

For more details about “nano” formatting option, please see newlib readme file, search for ‘enable-newlib-nano-formatted-io’ : <https://sourceware.org/newlib/README>

If this option is enabled, build system will use functions available in ROM, reducing the application binary size. Functions available in ROM run faster than functions which run from flash. Functions available in ROM can also run when flash instruction cache is disabled.

If you need 64-bit integer formatting support or C99 features, keep this option disabled.

NVS Contains:

- [CONFIG_NVS_ENCRYPTION](#)

CONFIG_NVS_ENCRYPTION

Enable NVS encryption

Found in: [Component config](#) > [NVS](#)

This option enables encryption for NVS. When enabled, AES-XTS is used to encrypt the complete NVS data, except the page headers. It requires XTS encryption keys to be stored in an encrypted partition. This means enabling flash encryption is a pre-requisite for this feature.

OpenSSL Contains:

- [CONFIG_OPENSSL_DEBUG](#)
- [CONFIG_OPENSSL_ASSERT](#)

CONFIG_OPENSSL_DEBUG

Enable OpenSSL debugging

Found in: [Component config](#) > [OpenSSL](#)

Enable OpenSSL debugging function.

If the option is enabled, “SSL_DEBUG” works.

CONFIG_OPENSSL_DEBUG_LEVEL

OpenSSL debugging level

Found in: [Component config](#) > [OpenSSL](#) > [CONFIG_OPENSSL_DEBUG](#)

OpenSSL debugging level.

Only function whose debugging level is higher than “OPENSSL_DEBUG_LEVEL” works.

For example: If OPENSSL_DEBUG_LEVEL = 2, you use function “SSL_DEBUG(1, “malloc failed”)” . Because $1 < 2$, it will not print.

CONFIG_OPENSSL_LOWLEVEL_DEBUG

Enable OpenSSL low-level module debugging

Found in: [Component config](#) > [OpenSSL](#) > [CONFIG_OPENSSL_DEBUG](#)

If the option is enabled, low-level module debugging function of OpenSSL is enabled, e.g. mbedtls internal debugging function.

CONFIG_OPENSSL_ASSERT

Select OpenSSL assert function

Found in: [Component config](#) > [OpenSSL](#)

OpenSSL function needs “assert” function to check if input parameters are valid.

If you want to use assert debugging function, “OPENSSL_DEBUG” should be enabled.

Available options:

- Do nothing (OPENSSL_ASSERT_DO_NOTHING)
Do nothing and “SSL_ASSERT” does not work.
- Check and exit (OPENSSL_ASSERT_EXIT)
Enable assert exiting, it will check and return error code.
- Show debugging message (OPENSSL_ASSERT_DEBUG)
Enable assert debugging, it will check and show debugging message.
- Show debugging message and exit (OPENSSL_ASSERT_DEBUG_EXIT)
Enable assert debugging and exiting, it will check, show debugging message and return error code.
- Show debugging message and block (OPENSSL_ASSERT_DEBUG_BLOCK)
Enable assert debugging and blocking, it will check, show debugging message and block by “while (1);” .

PThreads Contains:

- [CONFIG_PTHREAD_TASK_PRIO_DEFAULT](#)
- [CONFIG_PTHREAD_TASK_STACK_SIZE_DEFAULT](#)
- [CONFIG_PTHREAD_STACK_MIN](#)
- [CONFIG_PTHREAD_TASK_CORE_DEFAULT](#)
- [CONFIG_PTHREAD_TASK_NAME_DEFAULT](#)

CONFIG_PTHREAD_TASK_PRIO_DEFAULT

Default task priority

Found in: [Component config](#) > [PThreads](#)

Priority used to create new tasks with default pthread parameters.

CONFIG_PTHREAD_TASK_STACK_SIZE_DEFAULT

Default task stack size

Found in: [Component config](#) > [PThreads](#)

Stack size used to create new tasks with default pthread parameters.

CONFIG_PTHREAD_STACK_MIN

Minimum allowed pthread stack size

Found in: [Component config](#) > [PThreads](#)

Minimum allowed pthread stack size set in attributes passed to pthread_create

CONFIG_PTHREAD_TASK_CORE_DEFAULT

Default pthread core affinity

Found in: [Component config](#) > [PThreads](#)

The default core to which pthreads are pinned.

Available options:

- No affinity (PTHREAD_DEFAULT_CORE_NO_AFFINITY)
- Core 0 (PTHREAD_DEFAULT_CORE_0)
- Core 1 (PTHREAD_DEFAULT_CORE_1)

CONFIG_PTHREAD_TASK_NAME_DEFAULT

Default name of pthreads

Found in: [Component config](#) > [PThreads](#)

The default name of pthreads.

SPI Flash driver Contains:

- `CONFIG_SPI_FLASH_VERIFY_WRITE`
- `CONFIG_SPI_FLASH_ENABLE_COUNTERS`
- `CONFIG_SPI_FLASH_ROM_DRIVER_PATCH`
- `CONFIG_SPI_FLASH_DANGEROUS_WRITE`
- `CONFIG_SPI_FLASH_USE_LEGACY_IMPL`
- `CONFIG_SPI_FLASH_BYPASS_BLOCK_ERASE`
- `CONFIG_SPI_FLASH_YIELD_DURING_ERASE`
- `CONFIG_SPI_FLASH_SIZE_OVERRIDE`
- *Auto-detect flash chips*
- `CONFIG_SPI_FLASH_ENABLE_ENCRYPTED_READ_WRITE`

CONFIG_SPI_FLASH_VERIFY_WRITE

Verify SPI flash writes

Found in: [Component config](#) > [SPI Flash driver](#)

If this option is enabled, any time SPI flash is written then the data will be read back and verified. This can catch hardware problems with SPI flash, or flash which was not erased before verification.

CONFIG_SPI_FLASH_LOG_FAILED_WRITE

Log errors if verification fails

Found in: [Component config](#) > [SPI Flash driver](#) > [CONFIG_SPI_FLASH_VERIFY_WRITE](#)

If this option is enabled, if SPI flash write verification fails then a log error line will be written with the address, expected & actual values. This can be useful when debugging hardware SPI flash problems.

CONFIG_SPI_FLASH_WARN_SETTING_ZERO_TO_ONE

Log warning if writing zero bits to ones

Found in: [Component config](#) > [SPI Flash driver](#) > [CONFIG_SPI_FLASH_VERIFY_WRITE](#)

If this option is enabled, any SPI flash write which tries to set zero bits in the flash to ones will log a warning. Such writes will not result in the requested data appearing identically in flash once written, as SPI NOR flash can only set bits to one when an entire sector is erased. After erasing, individual bits can only be written from one to zero.

Note that some software (such as SPIFFS) which is aware of SPI NOR flash may write one bits as an optimisation, relying on the data in flash becoming a bitwise AND of the new data and any existing data. Such software will log spurious warnings if this option is enabled.

CONFIG_SPI_FLASH_ENABLE_COUNTERS

Enable operation counters

Found in: [Component config](#) > [SPI Flash driver](#)

This option enables the following APIs:

- `spi_flash_reset_counters`
- `spi_flash_dump_counters`
- `spi_flash_get_counters`

These APIs may be used to collect performance data for `spi_flash` APIs and to help understand behaviour of libraries which use SPI flash.

CONFIG_SPI_FLASH_ROM_DRIVER_PATCH

Enable SPI flash ROM driver patched functions

Found in: [Component config](#) > [SPI Flash driver](#)

Enable this flag to use patched versions of SPI flash ROM driver functions. This option is needed to write to flash on ESP32-D2WD, and any configuration where external SPI flash is connected to non-default pins.

CONFIG_SPI_FLASH_DANGEROUS_WRITE

Writing to dangerous flash regions

Found in: [Component config](#) > [SPI Flash driver](#)

SPI flash APIs can optionally abort or return a failure code if erasing or writing addresses that fall at the beginning of flash (covering the bootloader and partition table) or that overlap the app partition that contains the running app.

It is not recommended to ever write to these regions from an IDF app, and this check prevents logic errors or corrupted firmware memory from damaging these regions.

Note that this feature **does not** check calls to the `esp_rom_XXX` SPI flash ROM functions. These functions should not be called directly from IDF applications.

Available options:

- Aborts (`SPI_FLASH_DANGEROUS_WRITE_ABORTS`)
- Fails (`SPI_FLASH_DANGEROUS_WRITE_FAILS`)
- Allowed (`SPI_FLASH_DANGEROUS_WRITE_ALLOWED`)

CONFIG_SPI_FLASH_USE_LEGACY_IMPL

Use the legacy implementation before IDF v4.0

Found in: [Component config](#) > [SPI Flash driver](#)

The implementation of SPI flash has been greatly changed in IDF v4.0. Enable this option to use the legacy implementation.

CONFIG_SPI_FLASH_BYPASS_BLOCK_ERASE

Bypass a block erase and always do sector erase

Found in: [Component config](#) > [SPI Flash driver](#)

Some flash chips can have very high “max” erase times, especially for block erase (32KB or 64KB). This option allows to bypass “block erase” and always do sector erase commands. This will be much slower overall in most cases, but improves latency for other code to run.

CONFIG_SPI_FLASH_YIELD_DURING_ERASE

Enables yield operation during flash erase

Found in: [Component config](#) > [SPI Flash driver](#)

This allows to yield the CPUs between erase commands. Prevents starvation of other tasks.

CONFIG_SPI_FLASH_ERASE_YIELD_DURATION_MS

Duration of erasing to yield CPUs (ms)

Found in: [Component config](#) > [SPI Flash driver](#) > [CONFIG_SPI_FLASH_YIELD_DURING_ERASE](#)

If a duration of one erase command is large then it will yield CPUs after finishing a current command.

CONFIG_SPI_FLASH_ERASE_YIELD_TICKS

CPU release time (tick)

Found in: [Component config](#) > [SPI Flash driver](#) > [CONFIG_SPI_FLASH_YIELD_DURING_ERASE](#)

Defines how many ticks will be before returning to continue a erasing.

CONFIG_SPI_FLASH_SIZE_OVERRIDE

Override flash size in bootloader header by ESPTOOLPY_FLASHSIZE

Found in: Component config > SPI Flash driver

SPI Flash driver uses the flash size configured in bootloader header by default. Enable this option to override flash size with latest ESPTOOLPY_FLASHSIZE value from the app header if the size in the bootloader header is incorrect.

Auto-detect flash chips Contains:

- *CONFIG_SPI_FLASH_SUPPORT_ISSI_CHIP*
- *CONFIG_SPI_FLASH_SUPPORT_GD_CHIP*

CONFIG_SPI_FLASH_SUPPORT_ISSI_CHIP

ISSI

Found in: Component config > SPI Flash driver > Auto-detect flash chips

Enable this to support auto detection of ISSI chips if chip vendor not directly given by `chip_drv` member of the chip struct. This adds support for variant chips, however will extend detecting time.

CONFIG_SPI_FLASH_SUPPORT_GD_CHIP

GigaDevice

Found in: Component config > SPI Flash driver > Auto-detect flash chips

Enable this to support auto detection of GD (GigaDevice) chips if chip vendor not directly given by `chip_drv` member of the chip struct. If you are using Wrover modules, please don't disable this, otherwise your flash may not work in 4-bit mode.

This adds support for variant chips, however will extend detecting time and image size. Note that the default chip driver supports the GD chips with product ID 60H.

CONFIG_SPI_FLASH_ENABLE_ENCRYPTED_READ_WRITE

Enable encrypted partition read/write operations

Found in: Component config > SPI Flash driver

This option enables flash read/write operations to encrypted partition/s. This option is kept enabled irrespective of state of flash encryption feature. However, in case application is not using flash encryption feature and is in need of some additional memory from IRAM region (~1KB) then this config can be disabled.

SPIFFS Configuration Contains:

- *CONFIG_SPIFFS_MAX_PARTITIONS*
- *SPIFFS Cache Configuration*
- *CONFIG_SPIFFS_PAGE_CHECK*
- *CONFIG_SPIFFS_GC_MAX_RUNS*
- *CONFIG_SPIFFS_GC_STATS*
- *CONFIG_SPIFFS_PAGE_SIZE*
- *CONFIG_SPIFFS_OBJ_NAME_LEN*
- *CONFIG_SPIFFS_FOLLOW_SYMLINKS*
- *CONFIG_SPIFFS_USE_MAGIC*
- *CONFIG_SPIFFS_META_LENGTH*
- *CONFIG_SPIFFS_USE_MTIME*
- *CONFIG_SPIFFS_MTIME_WIDE_64_BITS*

- [Debug Configuration](#)

CONFIG_SPIFFS_MAX_PARTITIONS

Maximum Number of Partitions

Found in: [Component config](#) > [SPIFFS Configuration](#)

Define maximum number of partitions that can be mounted.

SPIFFS Cache Configuration Contains:

- [CONFIG_SPIFFS_CACHE](#)

CONFIG_SPIFFS_CACHE

Enable SPIFFS Cache

Found in: [Component config](#) > [SPIFFS Configuration](#) > [SPIFFS Cache Configuration](#)

Enables/disable memory read caching of nucleus file system operations.

CONFIG_SPIFFS_CACHE_WR

Enable SPIFFS Write Caching

Found in: [Component config](#) > [SPIFFS Configuration](#) > [SPIFFS Cache Configuration](#) > [CONFIG_SPIFFS_CACHE](#)

Enables memory write caching for file descriptors in hydrogen.

CONFIG_SPIFFS_CACHE_STATS

Enable SPIFFS Cache Statistics

Found in: [Component config](#) > [SPIFFS Configuration](#) > [SPIFFS Cache Configuration](#) > [CONFIG_SPIFFS_CACHE](#)

Enable/disable statistics on caching. Debug/test purpose only.

CONFIG_SPIFFS_PAGE_CHECK

Enable SPIFFS Page Check

Found in: [Component config](#) > [SPIFFS Configuration](#)

Always check header of each accessed page to ensure consistent state. If enabled it will increase number of reads from flash, especially if cache is disabled.

CONFIG_SPIFFS_GC_MAX_RUNS

Set Maximum GC Runs

Found in: [Component config](#) > [SPIFFS Configuration](#)

Define maximum number of GC runs to perform to reach desired free pages.

CONFIG_SPIFFS_GC_STATS

Enable SPIFFS GC Statistics

Found in: [Component config](#) > [SPIFFS Configuration](#)

Enable/disable statistics on gc. Debug/test purpose only.

CONFIG_SPIFFS_PAGE_SIZE

SPIFFS logical page size

Found in: [Component config](#) > [SPIFFS Configuration](#)

Logical page size of SPIFFS partition, in bytes. Must be multiple of flash page size (which is usually 256 bytes). Larger page sizes reduce overhead when storing large files, and improve filesystem performance when reading large files. Smaller page sizes reduce overhead when storing small (< page size) files.

CONFIG_SPIFFS_OBJ_NAME_LEN

Set SPIFFS Maximum Name Length

Found in: [Component config](#) > [SPIFFS Configuration](#)

Object name maximum length. Note that this length include the zero-termination character, meaning maximum string of characters can at most be SPIFFS_OBJ_NAME_LEN - 1.

SPIFFS_OBJ_NAME_LEN + SPIFFS_META_LENGTH should not exceed SPIFFS_PAGE_SIZE - 64.

CONFIG_SPIFFS_FOLLOW_SYMLINKS

Enable symbolic links for image creation

Found in: [Component config](#) > [SPIFFS Configuration](#)

If this option is enabled, symbolic links are taken into account during partition image creation.

CONFIG_SPIFFS_USE_MAGIC

Enable SPIFFS Filesystem Magic

Found in: [Component config](#) > [SPIFFS Configuration](#)

Enable this to have an identifiable spiffs filesystem. This will look for a magic in all sectors to determine if this is a valid spiffs system or not at mount time.

CONFIG_SPIFFS_USE_MAGIC_LENGTH

Enable SPIFFS Filesystem Length Magic

Found in: [Component config](#) > [SPIFFS Configuration](#) > [CONFIG_SPIFFS_USE_MAGIC](#)

If this option is enabled, the magic will also be dependent on the length of the filesystem. For example, a filesystem configured and formatted for 4 megabytes will not be accepted for mounting with a configuration defining the filesystem as 2 megabytes.

CONFIG_SPIFFS_META_LENGTH

Size of per-file metadata field

Found in: [Component config](#) > [SPIFFS Configuration](#)

This option sets the number of extra bytes stored in the file header. These bytes can be used in an application-specific manner. Set this to at least 4 bytes to enable support for saving file modification time.

SPIFFS_OBJ_NAME_LEN + SPIFFS_META_LENGTH should not exceed SPIFFS_PAGE_SIZE - 64.

CONFIG_SPIFFS_USE_MTIME

Save file modification time

Found in: [Component config](#) > [SPIFFS Configuration](#)

If enabled, then the first 4 bytes of per-file metadata will be used to store file modification time (mtime), accessible through stat/fstat functions. Modification time is updated when the file is opened.

CONFIG_SPIFFS_MTIME_WIDE_64_BITS

The time field occupies 64 bits in the image instead of 32 bits

Found in: [Component config](#) > [SPIFFS Configuration](#)

If this option is not set, the time field is 32 bits (up to 2106 year), otherwise it is 64 bits and make sure it matches SPIFFS_META_LENGTH. If the chip already has the spiffs image with the time field = 32 bits then this option cannot be applied in this case. Erase it first before using this option. To resolve the Y2K38 problem for the spiffs, use a toolchain with support time_t 64 bits (see SDK_TOOLCHAIN_SUPPORTS_TIME_WIDE_64_BITS).

Debug Configuration Contains:

- [CONFIG_SPIFFS_DBG](#)
- [CONFIG_SPIFFS_API_DBG](#)
- [CONFIG_SPIFFS_GC_DBG](#)
- [CONFIG_SPIFFS_CACHE_DBG](#)
- [CONFIG_SPIFFS_CHECK_DBG](#)
- [CONFIG_SPIFFS_TEST_VISUALISATION](#)

CONFIG_SPIFFS_DBG

Enable general SPIFFS debug

Found in: [Component config](#) > [SPIFFS Configuration](#) > [Debug Configuration](#)

Enabling this option will print general debug messages to the console.

CONFIG_SPIFFS_API_DBG

Enable SPIFFS API debug

Found in: [Component config](#) > [SPIFFS Configuration](#) > [Debug Configuration](#)

Enabling this option will print API debug messages to the console.

CONFIG_SPIFFS_GC_DBG

Enable SPIFFS Garbage Cleaner debug

Found in: [Component config](#) > [SPIFFS Configuration](#) > [Debug Configuration](#)

Enabling this option will print GC debug messages to the console.

CONFIG_SPIFFS_CACHE_DBG

Enable SPIFFS Cache debug

Found in: [Component config](#) > [SPIFFS Configuration](#) > [Debug Configuration](#)

Enabling this option will print cache debug messages to the console.

CONFIG_SPIFFS_CHECK_DBG

Enable SPIFFS Filesystem Check debug

Found in: [Component config](#) > [SPIFFS Configuration](#) > [Debug Configuration](#)

Enabling this option will print Filesystem Check debug messages to the console.

CONFIG_SPIFFS_TEST_VISUALISATION

Enable SPIFFS Filesystem Visualization

Found in: [Component config](#) > [SPIFFS Configuration](#) > [Debug Configuration](#)

Enable this option to enable SPIFFS_vis function in the API.

Unity unit testing library Contains:

- [CONFIG_UNITY_ENABLE_FLOAT](#)
- [CONFIG_UNITY_ENABLE_DOUBLE](#)
- [CONFIG_UNITY_ENABLE_COLOR](#)
- [CONFIG_UNITY_ENABLE_IDF_TEST_RUNNER](#)
- [CONFIG_UNITY_ENABLE_FIXTURE](#)
- [CONFIG_UNITY_ENABLE_BACKTRACE_ON_FAIL](#)

CONFIG_UNITY_ENABLE_FLOAT

Support for float type

Found in: [Component config](#) > [Unity unit testing library](#)

If not set, assertions on float arguments will not be available.

CONFIG_UNITY_ENABLE_DOUBLE

Support for double type

Found in: [Component config](#) > [Unity unit testing library](#)

If not set, assertions on double arguments will not be available.

CONFIG_UNITY_ENABLE_COLOR

Colorize test output

Found in: [Component config](#) > [Unity unit testing library](#)

If set, Unity will colorize test results using console escape sequences.

CONFIG_UNITY_ENABLE_IDF_TEST_RUNNER

Include ESP-IDF test registration/running helpers

Found in: [Component config](#) > [Unity unit testing library](#)

If set, then the following features will be available:

- TEST_CASE macro which performs automatic registration of test functions
- Functions to run registered test functions: `unity_run_all_tests`, `unity_run_tests_with_filter`, `unity_run_single_test_by_name`.
- Interactive menu which lists test cases and allows choosing the tests to be run, available via `unity_run_menu` function.

Disable if a different test registration mechanism is used.

CONFIG_UNITY_ENABLE_FIXTURE

Include Unity test fixture

Found in: Component config > Unity unit testing library

If set, unity_fixture.h header file and associated source files are part of the build. These provide an optional set of macros and functions to implement test groups.

CONFIG_UNITY_ENABLE_BACKTRACE_ON_FAIL

Print a backtrace when a unit test fails

Found in: Component config > Unity unit testing library

If set, the unity framework will print the backtrace information before jumping back to the test menu. The jumping is usually occurs in assert functions such as TEST_ASSERT, TEST_FAIL etc.

Virtual file system Contains:

- *CONFIG_VFS_SUPPRESS_SELECT_DEBUG_OUTPUT*
- *CONFIG_VFS_SUPPORT_TERMIOS*
- *Host File System I/O (Semihosting)*

CONFIG_VFS_SUPPRESS_SELECT_DEBUG_OUTPUT

Suppress select() related debug outputs

Found in: Component config > Virtual file system

Select() related functions might produce an inconveniently lot of debug outputs when one sets the default log level to DEBUG or higher. It is possible to suppress these debug outputs by enabling this option.

CONFIG_VFS_SUPPORT_TERMIOS

Add support for termios.h

Found in: Component config > Virtual file system

Disabling this option can save memory when the support for termios.h is not required.

Host File System I/O (Semihosting) Contains:

- *CONFIG_SEMIHOSTFS_MAX_MOUNT_POINTS*
- *CONFIG_SEMIHOSTFS_HOST_PATH_MAX_LEN*

CONFIG_SEMIHOSTFS_MAX_MOUNT_POINTS

Maximum number of the host filesystem mount points

Found in: Component config > Virtual file system > Host File System I/O (Semihosting)

Define maximum number of host filesystem mount points.

CONFIG_SEMIHOSTFS_HOST_PATH_MAX_LEN

Maximum path length for the host base directory

Found in: Component config > Virtual file system > Host File System I/O (Semihosting)

Define maximum path length for the host base directory which is to be mounted. If host path passed to esp_vfs_semihost_register() is longer than this value it will be truncated.

Wear Levelling Contains:

- [CONFIG_WL_SECTOR_SIZE](#)
- [CONFIG_WL_SECTOR_MODE](#)

CONFIG_WL_SECTOR_SIZE

Wear Levelling library sector size

Found in: [Component config](#) > [Wear Levelling](#)

Sector size used by wear levelling library. You can set default sector size or size that will fit to the flash device sector size.

With sector size set to 4096 bytes, wear levelling library is more efficient. However if FAT filesystem is used on top of wear levelling library, it will need more temporary storage: 4096 bytes for each mounted filesystem and 4096 bytes for each opened file.

With sector size set to 512 bytes, wear levelling library will perform more operations with flash memory, but less RAM will be used by FAT filesystem library (512 bytes for the filesystem and 512 bytes for each file opened).

Available options:

- 512 (WL_SECTOR_SIZE_512)
- 4096 (WL_SECTOR_SIZE_4096)

CONFIG_WL_SECTOR_MODE

Sector store mode

Found in: [Component config](#) > [Wear Levelling](#)

Specify the mode to store data into flash:

- In Performance mode a data will be stored to the RAM and then stored back to the flash. Compared to the Safety mode, this operation is faster, but if power will be lost when erase sector operation is in progress, then the data from complete flash device sector will be lost.
- In Safety mode data from complete flash device sector will be read from flash, modified, and then stored back to flash. Compared to the Performance mode, this operation is slower, but if power is lost during erase sector operation, then the data from full flash device sector will not be lost.

Available options:

- Performance (WL_SECTOR_MODE_PERF)
- Safety (WL_SECTOR_MODE_SAFE)

Wi-Fi Provisioning Manager Contains:

- [CONFIG_WIFI_PROV_SCAN_MAX_ENTRIES](#)
- [CONFIG_WIFI_PROV_AUTOSTOP_TIMEOUT](#)

CONFIG_WIFI_PROV_SCAN_MAX_ENTRIES

Max Wi-Fi Scan Result Entries

Found in: [Component config](#) > [Wi-Fi Provisioning Manager](#)

This sets the maximum number of entries of Wi-Fi scan results that will be kept by the provisioning manager

CONFIG_WIFI_PROV_AUTOSTOP_TIMEOUT

Provisioning auto-stop timeout

Found in: [Component config](#) > [Wi-Fi Provisioning Manager](#)

Time (in seconds) after which the Wi-Fi provisioning manager will auto-stop after connecting to a Wi-Fi network successfully.

Supplicant Contains:

- [CONFIG_WPA_MBEDTLS_CRYPTO](#)
- [CONFIG_WPA_TLS_V12](#)
- [CONFIG_WPA_WPS_STRICT](#)
- [CONFIG_WPA_DEBUG_PRINT](#)

CONFIG_WPA_MBEDTLS_CRYPTO

Use MbedTLS crypto API' s

Found in: [Component config](#) > [Supplicant](#)

Select this option to use MbedTLS crypto API' s which utilize hardware acceleration.

CONFIG_WPA_TLS_V12

Enable TLS v1.2

Found in: [Component config](#) > [Supplicant](#)

Select this to enable TLS v1.2 for WPA2-Enterprise Authentication.

CONFIG_WPA_WPS_STRICT

Strictly validate all WPS attributes

Found in: [Component config](#) > [Supplicant](#)

Select this option to enable validate each WPS attribute rigorously. Disabling this add the workarounds with various APs. Enabling this may cause inter operability issues with some APs.

CONFIG_WPA_DEBUG_PRINT

Print debug messages from WPA Supplicant

Found in: [Component config](#) > [Supplicant](#)

Select this option to print logging information from WPA supplicant, this includes handshake information and key hex dumps depending on the project logging level.

Enabling this could increase the build size ~60kb depending on the project logging level.

Compatibility options

Contains:

- [CONFIG_LEGACY_INCLUDE_COMMON_HEADERS](#)

CONFIG_LEGACY_INCLUDE_COMMON_HEADERS

Include headers across components as before IDF v4.0

Found in: *Compatibility options*

Soc, esp32, and driver components, the most common components. Some header of these components are included implicitly by headers of other components before IDF v4.0. It's not required for high-level components, but still included through long header chain everywhere.

This is harmful to the modularity. So it's changed in IDF v4.0.

You can still include these headers in a legacy way until it is totally deprecated by enable this option.

Deprecated options and their replacements

- CONFIG_A2DP_ENABLE ([CONFIG_BT_A2DP_ENABLE](#))
- CONFIG_ADC2_DISABLE_DAC ([CONFIG_ADC_DISABLE_DAC](#))
- CONFIG_APP_ANTI_ROLLBACK ([CONFIG_BOOTLOADER_APP_ANTI_ROLLBACK](#))
- CONFIG_APP_ROLLBACK_ENABLE ([CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE](#))
- CONFIG_APP_SECURE_VERSION ([CONFIG_BOOTLOADER_APP_SECURE_VERSION](#))
- CONFIG_APP_SECURE_VERSION_SIZE_EFUSE_FIELD ([CONFIG_BOOTLOADER_APP_SEC_VER_SIZE_EFUSE_FIELD](#))
- CONFIG_BLE_ACTIVE_SCAN_REPORT_ADV_SCAN_RSP_INDIVIDUALLY ([CONFIG_BT_BLE_ACT_SCAN_REP_ADV_SCAN](#)) (CON-
- CONFIG_BLE_ADV_REPORT_DISCARD_THRSHOLD ([CONFIG_BTDM_BLE_ADV_REPORT_DISCARD_THRSHOLD](#))
- CONFIG_BLE_ADV_REPORT_FLOW_CONTROL_NUM ([CONFIG_BTDM_BLE_ADV_REPORT_FLOW_CTRL_NUM](#))
- CONFIG_BLE_ADV_REPORT_FLOW_CONTROL_SUPPORTED ([CONFIG_BTDM_BLE_ADV_REPORT_FLOW_CTRL_SUPP](#)) (CON-
- CONFIG_BLE_ESTABLISH_LINK_CONNECTION_TIMEOUT ([CONFIG_BT_BLE_ESTAB_LINK_CONN_TOUT](#)) (CON-
- CONFIG_BLE_HOST_QUEUE_CONGESTION_CHECK ([CONFIG_BT_BLE_HOST_QUEUE_CONG_CHECK](#))
- CONFIG_BLE_MESH_GATT_PROXY ([CONFIG_BLE_MESH_GATT_PROXY_SERVER](#))
- CONFIG_BLE_MESH_SCAN_DUPLICATE_EN ([CONFIG_BTDM_BLE_MESH_SCAN_DUPL_EN](#))
- CONFIG_BLE_SCAN_DUPLICATE ([CONFIG_BTDM_BLE_SCAN_DUPL](#))
- CONFIG_BLE_SMP_ENABLE ([CONFIG_BT_BLE_SMP_ENABLE](#))
- CONFIG_BLUEDROID_MEM_DEBUG ([CONFIG_BT_BLUEDROID_MEM_DEBUG](#))
- CONFIG_BLUEDROID_PINNED_TO_CORE_CHOICE ([CONFIG_BT_BLUEDROID_PINNED_TO_CORE_CHOICE](#))
 - CONFIG_BLUEDROID_PINNED_TO_CORE_0
 - CONFIG_BLUEDROID_PINNED_TO_CORE_1
- CONFIG_BROWNOUT_DET ([CONFIG_ESP32_BROWNOUT_DET](#))
- CONFIG_BROWNOUT_DET_LVL_SEL ([CONFIG_ESP32_BROWNOUT_DET_LVL_SEL](#))
 - CONFIG_BROWNOUT_DET_LVL_SEL_0
 - CONFIG_BROWNOUT_DET_LVL_SEL_1
 - CONFIG_BROWNOUT_DET_LVL_SEL_2
 - CONFIG_BROWNOUT_DET_LVL_SEL_3
 - CONFIG_BROWNOUT_DET_LVL_SEL_4
 - CONFIG_BROWNOUT_DET_LVL_SEL_5
 - CONFIG_BROWNOUT_DET_LVL_SEL_6
 - CONFIG_BROWNOUT_DET_LVL_SEL_7
- CONFIG_BTC_TASK_STACK_SIZE ([CONFIG_BT_BTC_TASK_STACK_SIZE](#))
- CONFIG_BTDM_CONTROLLER_BLE_MAX_CONN ([CONFIG_BTDM_CTRL_BLE_MAX_CONN](#))
- CONFIG_BTDM_CONTROLLER_BR_EDR_MAX_ACL_CONN ([CONFIG_BTDM_CTRL_BR_EDR_MAX_ACL_CONN](#)) (CON-
- CONFIG_BTDM_CONTROLLER_BR_EDR_MAX_SYNC_CONN ([CONFIG_BTDM_CTRL_BR_EDR_MAX_SYNC_CONN](#)) (CON-
- CONFIG_BTDM_CONTROLLER_FULL_SCAN_SUPPORTED ([CONFIG_BTDM_CTRL_FULL_SCAN_SUPPORTED](#)) (CON-
- CONFIG_BTDM_CONTROLLER_HCI_MODE_CHOICE ([CONFIG_BTDM_CTRL_HCI_MODE_CHOICE](#))

- CONFIG_BTDM_CONTROLLER_HCI_MODE_VHCI
- CONFIG_BTDM_CONTROLLER_HCI_MODE_UART_H4
- **CONFIG_BTDM_CONTROLLER_MODE** ([CONFIG_BTDM_CTRL_MODE](#))
 - CONFIG_BTDM_CONTROLLER_MODE_BLE_ONLY
 - CONFIG_BTDM_CONTROLLER_MODE_BR_EDR_ONLY
 - CONFIG_BTDM_CONTROLLER_MODE_BTDM
- CONFIG_BTDM_CONTROLLER_MODEM_SLEEP ([CONFIG_BTDM_MODEM_SLEEP](#))
- CONFIG_BTDM_CONTROLLER_PINNED_TO_CORE_CHOICE ([CONFIG_BTDM_CTRL_PINNED_TO_CORE_CHOICE](#))
- CONFIG_BTU_TASK_STACK_SIZE ([CONFIG_BT_BTU_TASK_STACK_SIZE](#))
- CONFIG_CLASSIC_BT_ENABLED ([CONFIG_BT_CLASSIC_ENABLED](#))
- CONFIG_COMPATIBLE_PRE_V2_1_BOOTLOADERS ([CONFIG_ESP32_COMPATIBLE_PRE_V2_1_BOOTLOADERS](#))
- **CONFIG_CONSOLE_UART** ([CONFIG_ESP_CONSOLE_UART](#))
 - CONFIG_CONSOLE_UART_DEFAULT
 - CONFIG_CONSOLE_UART_CUSTOM
 - CONFIG_CONSOLE_UART_NONE
- CONFIG_CONSOLE_UART_BAUDRATE ([CONFIG_ESP_CONSOLE_UART_BAUDRATE](#))
- CONFIG_CONSOLE_UART_RX_GPIO ([CONFIG_ESP_CONSOLE_UART_RX_GPIO](#))
- CONFIG_CONSOLE_UART_TX_GPIO ([CONFIG_ESP_CONSOLE_UART_TX_GPIO](#))
- CONFIG_CXX_EXCEPTIONS ([CONFIG_COMPILER_CXX_EXCEPTIONS](#))
- CONFIG_CXX_EXCEPTIONS_EMG_POOL_SIZE ([CONFIG_COMPILER_CXX_EXCEPTIONS_EMG_POOL_SIZE](#))
- CONFIG_DISABLE_BASIC_ROM_CONSOLE ([CONFIG_ESP32_DISABLE_BASIC_ROM_CONSOLE](#))
- CONFIG_DISABLE_GCC8_WARNINGS ([CONFIG_COMPILER_DISABLE_GCC8_WARNINGS](#))
- CONFIG_DUPLICATE_SCAN_CACHE_SIZE ([CONFIG_BTDM_SCAN_DUPL_CACHE_SIZE](#))
- CONFIG_EFUSE_SECURE_VERSION_EMULATE ([CONFIG_BOOTLOADER_EFUSE_SECURE_VERSION_EMULATE](#))
- CONFIG_ENABLE_STATIC_TASK_CLEAN_UP_HOOK ([CONFIG_FREERTOS_ENABLE_STATIC_TASK_CLEAN_UP](#))
- **CONFIG_ESP32_APPTRACE_DESTINATION** ([CONFIG_APPTRACE_DESTINATION](#))
 - CONFIG_ESP32_APPTRACE_DEST_TRAX
 - CONFIG_ESP32_APPTRACE_DEST_NONE
- CONFIG_ESP32_APPTRACE_ONPANIC_HOST_FLUSH_TMO ([CONFIG_APPTRACE_ONPANIC_HOST_FLUSH_TMO](#))
- CONFIG_ESP32_APPTRACE_PENDING_DATA_SIZE_MAX ([CONFIG_APPTRACE_PENDING_DATA_SIZE_MAX](#))
- CONFIG_ESP32_APPTRACE_POSTMORTEM_FLUSH_TRAX_THRESH ([CONFIG_APPTRACE_POSTMORTEM_FLUSH_THRESH](#))
- CONFIG_ESP32_GCOV_ENABLE ([CONFIG_APPTRACE_GCOV_ENABLE](#))
- CONFIG_ESP32_PTHREAD_STACK_MIN ([CONFIG_PTHREAD_STACK_MIN](#))
- CONFIG_ESP32_PTHREAD_TASK_NAME_DEFAULT ([CONFIG_PTHREAD_TASK_NAME_DEFAULT](#))
- CONFIG_ESP32_PTHREAD_TASK_PRIO_DEFAULT ([CONFIG_PTHREAD_TASK_PRIO_DEFAULT](#))
- CONFIG_ESP32_PTHREAD_TASK_STACK_SIZE_DEFAULT ([CONFIG_PTHREAD_TASK_STACK_SIZE_DEFAULT](#))
- **CONFIG_ESP32_RTC_CLOCK_SOURCE** ([CONFIG_ESP32_RTC_CLK_SRC](#))
 - CONFIG_ESP32_RTC_CLOCK_SOURCE_INTERNAL_RC
 - CONFIG_ESP32_RTC_CLOCK_SOURCE_EXTERNAL_CRYSTAL
 - CONFIG_ESP32_RTC_CLOCK_SOURCE_EXTERNAL_OSC
 - CONFIG_ESP32_RTC_CLOCK_SOURCE_INTERNAL_8MD256
- CONFIG_ESP32_RTC_EXTERNAL_CRYSTAL_ADDITIONAL_CURRENT ([CONFIG_ESP32_RTC_EXT_CRYST_ADDIT_CURRENT](#))
- CONFIG_ESP_GRATUITOUS_ARP ([CONFIG_LWIP_ESP_GRATUITOUS_ARP](#))
- CONFIG_ESP_TCP_KEEP_CONNECTION_WHEN_IP_CHANGES ([CONFIG_LWIP_TCP_KEEP_CONNECTION_WHEN_IP_CHANGES](#))
- CONFIG_EVENT_LOOP_PROFILING ([CONFIG_ESP_EVENT_LOOP_PROFILING](#))
- CONFIG_FLASH_ENCRYPTION_ENABLED ([CONFIG_SECURE_FLASH_ENC_ENABLED](#))
- CONFIG_FLASH_ENCRYPTION_UART_BOOTLOADER_ALLOW_CACHE ([CONFIG_SECURE_FLASH_UART_BOOTLOADER_ALLOW_CACHE](#))
- CONFIG_FLASH_ENCRYPTION_UART_BOOTLOADER_ALLOW_DECRYPT ([CONFIG_SECURE_FLASH_UART_BOOTLOADER_ALLOW_DEC](#))
- CONFIG_FLASH_ENCRYPTION_UART_BOOTLOADER_ALLOW_ENCRYPT ([CONFIG_SECURE_FLASH_UART_BOOTLOADER_ALLOW_ENC](#))
- CONFIG_GARP_TMR_INTERVAL ([CONFIG_LWIP_GARP_TMR_INTERVAL](#))

- CONFIG_GATTC_CACHE_NVS_FLASH (*CONFIG_BT_GATTC_CACHE_NVS_FLASH*)
- CONFIG_GATTC_ENABLE (*CONFIG_BT_GATTC_ENABLE*)
- CONFIG_GATTS_ENABLE (*CONFIG_BT_GATTS_ENABLE*)
- CONFIG_GDBSTUB_MAX_TASKS (*CONFIG_ESP_GDBSTUB_MAX_TASKS*)
- CONFIG_GDBSTUB_SUPPORT_TASKS (*CONFIG_ESP_GDBSTUB_SUPPORT_TASKS*)
- **CONFIG_HFP_AUDIO_DATA_PATH** (*CONFIG_BT_HFP_AUDIO_DATA_PATH*)
 - CONFIG_HFP_AUDIO_DATA_PATH_PCM
 - CONFIG_HFP_AUDIO_DATA_PATH_HCI
- CONFIG_HFP_ENABLE (*CONFIG_BT_HFP_ENABLE*)
- **CONFIG_HFP_ROLE** (*CONFIG_BT_HFP_ROLE*)
 - CONFIG_HFP_CLIENT_ENABLE
 - CONFIG_HFP_AG_ENABLE
- CONFIG_INT_WDT (*CONFIG_ESP_INT_WDT*)
- CONFIG_INT_WDT_CHECK_CPU1 (*CONFIG_ESP_INT_WDT_CHECK_CPU1*)
- CONFIG_INT_WDT_TIMEOUT_MS (*CONFIG_ESP_INT_WDT_TIMEOUT_MS*)
- CONFIG_IPC_TASK_STACK_SIZE (*CONFIG_ESP_IPC_TASK_STACK_SIZE*)
- CONFIG_L2_TO_L3_COPY (*CONFIG_LWIP_L2_TO_L3_COPY*)
- CONFIG_MAIN_TASK_STACK_SIZE (*CONFIG_ESP_MAIN_TASK_STACK_SIZE*)
- CONFIG_MAKE_WARN_UNDEFINED_VARIABLES (*CONFIG_SDK_MAKE_WARN_UNDEFINED_VARIABLES*)
- CONFIG_MB_CONTROLLER_NOTIFY_QUEUE_SIZE (*CONFIG_FMB_CONTROLLER_NOTIFY_QUEUE_SIZE*)
- CONFIG_MB_CONTROLLER_NOTIFY_TIMEOUT (*CONFIG_FMB_CONTROLLER_NOTIFY_TIMEOUT*)
- CONFIG_MB_CONTROLLER_SLAVE_ID (*CONFIG_FMB_CONTROLLER_SLAVE_ID*)
- CONFIG_MB_CONTROLLER_SLAVE_ID_SUPPORT (*CONFIG_FMB_CONTROLLER_SLAVE_ID_SUPPORT*)
- CONFIG_MB_CONTROLLER_STACK_SIZE (*CONFIG_FMB_CONTROLLER_STACK_SIZE*)
- CONFIG_MB_EVENT_QUEUE_TIMEOUT (*CONFIG_FMB_EVENT_QUEUE_TIMEOUT*)
- CONFIG_MB_MASTER_DELAY_MS_CONVERT (*CONFIG_FMB_MASTER_DELAY_MS_CONVERT*)
- CONFIG_MB_MASTER_TIMEOUT_MS_RESPOND (*CONFIG_FMB_MASTER_TIMEOUT_MS_RESPOND*)
- CONFIG_MB_QUEUE_LENGTH (*CONFIG_FMB_QUEUE_LENGTH*)
- CONFIG_MB_SERIAL_BUF_SIZE (*CONFIG_FMB_SERIAL_BUF_SIZE*)
- CONFIG_MB_SERIAL_TASK_PRIO (*CONFIG_FMB_SERIAL_TASK_PRIO*)
- CONFIG_MB_SERIAL_TASK_STACK_SIZE (*CONFIG_FMB_SERIAL_TASK_STACK_SIZE*)
- CONFIG_MB_TIMER_GROUP (*CONFIG_FMB_TIMER_GROUP*)
- CONFIG_MB_TIMER_INDEX (*CONFIG_FMB_TIMER_INDEX*)
- CONFIG_MB_TIMER_PORT_ENABLED (*CONFIG_FMB_TIMER_PORT_ENABLED*)
- CONFIG_MESH_DUPLICATE_SCAN_CACHE_SIZE (*CONFIG_BTDM_MESH_DUPL_SCAN_CACHE_SIZE*)
- CONFIG_MONITOR_BAUD_OTHER_VAL (*CONFIG_ESPTOOLPY_MONITOR_BAUD_OTHER_VAL*)
- CONFIG_NIMBLE_ACL_BUF_COUNT (*CONFIG_BT_NIMBLE_ACL_BUF_COUNT*)
- CONFIG_NIMBLE_ACL_BUF_SIZE (*CONFIG_BT_NIMBLE_ACL_BUF_SIZE*)
- CONFIG_NIMBLE_ATT_PREFERRED_MTU (*CONFIG_BT_NIMBLE_ATT_PREFERRED_MTU*)
- CONFIG_NIMBLE_CRYPTOSTACK_MBEDTLS (*CONFIG_BT_NIMBLE_CRYPTOSTACK_MBEDTLS*)
- CONFIG_NIMBLE_DEBUG (*CONFIG_BT_NIMBLE_DEBUG*)
- CONFIG_NIMBLE_GAP_DEVICE_NAME_MAX_LEN (*CONFIG_BT_NIMBLE_GAP_DEVICE_NAME_MAX_LEN*)
- CONFIG_NIMBLE_HCI_EVT_BUF_SIZE (*CONFIG_BT_NIMBLE_HCI_EVT_BUF_SIZE*)
- CONFIG_NIMBLE_HCI_EVT_HI_BUF_COUNT (*CONFIG_BT_NIMBLE_HCI_EVT_HI_BUF_COUNT*)
- CONFIG_NIMBLE_HCI_EVT_LO_BUF_COUNT (*CONFIG_BT_NIMBLE_HCI_EVT_LO_BUF_COUNT*)
- CONFIG_NIMBLE_HS_FLOW_CTRL (*CONFIG_BT_NIMBLE_HS_FLOW_CTRL*)
- CONFIG_NIMBLE_HS_FLOW_CTRL_ITVL (*CONFIG_BT_NIMBLE_HS_FLOW_CTRL_ITVL*)
- CONFIG_NIMBLE_HS_FLOW_CTRL_THRESH (*CONFIG_BT_NIMBLE_HS_FLOW_CTRL_THRESH*)
- CONFIG_NIMBLE_HS_FLOW_CTRL_TX_ON_DISCONNECT (*CONFIG_BT_NIMBLE_HS_FLOW_CTRL_TX_ON_DISCONNECT*)
- CONFIG_NIMBLE_L2CAP_COC_MAX_NUM (*CONFIG_BT_NIMBLE_L2CAP_COC_MAX_NUM*)
- CONFIG_NIMBLE_MAX_BONDS (*CONFIG_BT_NIMBLE_MAX_BONDS*)
- CONFIG_NIMBLE_MAX_CCCDS (*CONFIG_BT_NIMBLE_MAX_CCCDS*)
- CONFIG_NIMBLE_MAX_CONNECTIONS (*CONFIG_BT_NIMBLE_MAX_CONNECTIONS*)
- **CONFIG_NIMBLE_MEM_ALLOC_MODE** (*CONFIG_BT_NIMBLE_MEM_ALLOC_MODE*)
 - CONFIG_NIMBLE_MEM_ALLOC_MODE_INTERNAL
 - CONFIG_NIMBLE_MEM_ALLOC_MODE_EXTERNAL
 - CONFIG_NIMBLE_MEM_ALLOC_MODE_DEFAULT

- CONFIG_NIMBLE_MESH (*CONFIG_BT_NIMBLE_MESH*)
- CONFIG_NIMBLE_MESH_DEVICE_NAME (*CONFIG_BT_NIMBLE_MESH_DEVICE_NAME*)
- CONFIG_NIMBLE_MESH_FRIEND (*CONFIG_BT_NIMBLE_MESH_FRIEND*)
- CONFIG_NIMBLE_MESH_GATT_PROXY (*CONFIG_BT_NIMBLE_MESH_GATT_PROXY*)
- CONFIG_NIMBLE_MESH_LOW_POWER (*CONFIG_BT_NIMBLE_MESH_LOW_POWER*)
- CONFIG_NIMBLE_MESH_PB_ADV (*CONFIG_BT_NIMBLE_MESH_PB_ADV*)
- CONFIG_NIMBLE_MESH_PB_GATT (*CONFIG_BT_NIMBLE_MESH_PB_GATT*)
- CONFIG_NIMBLE_MESH_PROV (*CONFIG_BT_NIMBLE_MESH_PROV*)
- CONFIG_NIMBLE_MESH_PROXY (*CONFIG_BT_NIMBLE_MESH_PROXY*)
- CONFIG_NIMBLE_MESH_RELAY (*CONFIG_BT_NIMBLE_MESH_RELAY*)
- CONFIG_NIMBLE_MSYS1_BLOCK_COUNT (*CONFIG_BT_NIMBLE_MSYS1_BLOCK_COUNT*)
- CONFIG_NIMBLE_NVS_PERSIST (*CONFIG_BT_NIMBLE_NVS_PERSIST*)
- **CONFIG_NIMBLE_PINNED_TO_CORE_CHOICE (*CONFIG_BT_NIMBLE_PINNED_TO_CORE_CHOICE*)**
 - CONFIG_NIMBLE_PINNED_TO_CORE_0
 - CONFIG_NIMBLE_PINNED_TO_CORE_1
- CONFIG_NIMBLE_ROLE_BROADCASTER (*CONFIG_BT_NIMBLE_ROLE_BROADCASTER*)
- CONFIG_NIMBLE_ROLE_CENTRAL (*CONFIG_BT_NIMBLE_ROLE_CENTRAL*)
- CONFIG_NIMBLE_ROLE_OBSERVER (*CONFIG_BT_NIMBLE_ROLE_OBSERVER*)
- CONFIG_NIMBLE_ROLE_PERIPHERAL (*CONFIG_BT_NIMBLE_ROLE_PERIPHERAL*)
- CONFIG_NIMBLE_RPA_TIMEOUT (*CONFIG_BT_NIMBLE_RPA_TIMEOUT*)
- CONFIG_NIMBLE_SM_LEGACY (*CONFIG_BT_NIMBLE_SM_LEGACY*)
- CONFIG_NIMBLE_SM_SC (*CONFIG_BT_NIMBLE_SM_SC*)
- CONFIG_NIMBLE_SM_SC_DEBUG_KEYS (*CONFIG_BT_NIMBLE_SM_SC_DEBUG_KEYS*)
- CONFIG_NIMBLE_SVC_GAP_APPEARANCE (*CONFIG_BT_NIMBLE_SVC_GAP_APPEARANCE*)
- CONFIG_NIMBLE_SVC_GAP_DEVICE_NAME (*CONFIG_BT_NIMBLE_SVC_GAP_DEVICE_NAME*)
- CONFIG_NIMBLE_TASK_STACK_SIZE (*CONFIG_BT_NIMBLE_TASK_STACK_SIZE*)
- CONFIG_NO_BLOBS (*CONFIG_ESP32_NO_BLOBS*)
- **CONFIG_OPTIMIZATION_ASSERTION_LEVEL (*CONFIG_COMPILER_OPTIMIZATION_ASSERTION_LEVEL*)**
 - CONFIG_OPTIMIZATION_ASSERTIONS_ENABLED
 - CONFIG_OPTIMIZATION_ASSERTIONS_SILENT
 - CONFIG_OPTIMIZATION_ASSERTIONS_DISABLED
- **CONFIG_OPTIMIZATION_COMPILER (*CONFIG_COMPILER_OPTIMIZATION*)**
 - CONFIG_COMPILER_OPTIMIZATION_LEVEL_DEBUG
 - CONFIG_COMPILER_OPTIMIZATION_LEVEL_RELEASE
- CONFIG_POST_EVENTS_FROM_IRAM_ISR (*CONFIG_ESP_EVENT_POST_FROM_IRAM_ISR*)
- CONFIG_POST_EVENTS_FROM_ISR (*CONFIG_ESP_EVENT_POST_FROM_ISR*)
- CONFIG_PPP_CHAP_SUPPORT (*CONFIG_LWIP_PPP_CHAP_SUPPORT*)
- CONFIG_PPP_DEBUG_ON (*CONFIG_LWIP_PPP_DEBUG_ON*)
- CONFIG_PPP_MPPE_SUPPORT (*CONFIG_LWIP_PPP_MPPE_SUPPORT*)
- CONFIG_PPP_MSCHAP_SUPPORT (*CONFIG_LWIP_PPP_MSCHAP_SUPPORT*)
- CONFIG_PPP_NOTIFY_PHASE_SUPPORT (*CONFIG_LWIP_PPP_NOTIFY_PHASE_SUPPORT*)
- CONFIG_PPP_PAP_SUPPORT (*CONFIG_LWIP_PPP_PAP_SUPPORT*)
- CONFIG_PPP_SUPPORT (*CONFIG_LWIP_PPP_SUPPORT*)
- CONFIG_PYTHON (*CONFIG_SDK_PYTHON*)
- CONFIG_REDUCE_PHY_TX_POWER (*CONFIG_ESP32_REDUCE_PHY_TX_POWER*)
- CONFIG_SMP_SLAVE_CON_PARAMS_UPD_ENABLE (*CONFIG_BT_SMP_SLAVE_CON_PARAMS_UPD_ENABLE*)
- CONFIG_SPIRAM_SUPPORT (*CONFIG_ESP32_SPIRAM_SUPPORT*)
- **CONFIG_SPI_FLASH_WRITING_DANGEROUS_REGIONS (*CONFIG_SPI_FLASH_DANGEROUS_WRITE*)**
 - CONFIG_SPI_FLASH_WRITING_DANGEROUS_REGIONS_ABORTS
 - CONFIG_SPI_FLASH_WRITING_DANGEROUS_REGIONS_FAILS
 - CONFIG_SPI_FLASH_WRITING_DANGEROUS_REGIONS_ALLOWED
- **CONFIG_STACK_CHECK_MODE (*CONFIG_COMPILER_STACK_CHECK_MODE*)**
 - CONFIG_STACK_CHECK_NONE
 - CONFIG_STACK_CHECK_NORM
 - CONFIG_STACK_CHECK_STRONG

- CONFIG_STACK_CHECK_ALL
- CONFIG_SUPPORT_STATIC_ALLOCATION (*CONFIG_FREERTOS_SUPPORT_STATIC_ALLOCATION*)
- CONFIG_SUPPORT_TERMIOS (*CONFIG_VFS_SUPPORT_TERMIOS*)
- CONFIG_SUPPRESS_SELECT_DEBUG_OUTPUT (*CONFIG_VFS_SUPPRESS_SELECT_DEBUG_OUTPUT*)
- CONFIG_SW_COEXIST_ENABLE (*CONFIG_ESP32_WIFI_SW_COEXIST_ENABLE*)
- CONFIG_SYSTEM_EVENT_QUEUE_SIZE (*CONFIG_ESP_SYSTEM_EVENT_QUEUE_SIZE*)
- CONFIG_SYSTEM_EVENT_TASK_STACK_SIZE (*CONFIG_ESP_SYSTEM_EVENT_TASK_STACK_SIZE*)
- CONFIG_TASK_WDT (*CONFIG_ESP_TASK_WDT*)
- CONFIG_TASK_WDT_CHECK_IDLE_TASK_CPU0 (*CONFIG_ESP_TASK_WDT_CHECK_IDLE_TASK_CPU0*)
- CONFIG_TASK_WDT_CHECK_IDLE_TASK_CPU1 (*CONFIG_ESP_TASK_WDT_CHECK_IDLE_TASK_CPU1*)
- CONFIG_TASK_WDT_PANIC (*CONFIG_ESP_TASK_WDT_PANIC*)
- CONFIG_TASK_WDT_TIMEOUT_S (*CONFIG_ESP_TASK_WDT_TIMEOUT_S*)
- CONFIG_TCPIP_RECVMBOX_SIZE (*CONFIG_LWIP_TCPIP_RECVMBOX_SIZE*)
- CONFIG_TCPIP_TASK_STACK_SIZE (*CONFIG_LWIP_TCPIP_TASK_STACK_SIZE*)
- CONFIG_TCP_MAXRTX (*CONFIG_LWIP_TCP_MAXRTX*)
- CONFIG_TCP_MSL (*CONFIG_LWIP_TCP_MSL*)
- CONFIG_TCP_MSS (*CONFIG_LWIP_TCP_MSS*)
- **CONFIG_TCP_OVERSIZE** (*CONFIG_LWIP_TCP_OVERSIZE*)
 - CONFIG_TCP_OVERSIZE_MSS
 - CONFIG_TCP_OVERSIZE_QUARTER_MSS
 - CONFIG_TCP_OVERSIZE_DISABLE
- CONFIG_TCP_QUEUE_OOSEQ (*CONFIG_LWIP_TCP_QUEUE_OOSEQ*)
- CONFIG_TCP_RECVMBOX_SIZE (*CONFIG_LWIP_TCP_RECVMBOX_SIZE*)
- CONFIG_TCP_SND_BUF_DEFAULT (*CONFIG_LWIP_TCP_SND_BUF_DEFAULT*)
- CONFIG_TCP_SYNMAXRTX (*CONFIG_LWIP_TCP_SYNMAXRTX*)
- CONFIG_TCP_WND_DEFAULT (*CONFIG_LWIP_TCP_WND_DEFAULT*)
- CONFIG_TIMER_QUEUE_LENGTH (*CONFIG_FREERTOS_TIMER_QUEUE_LENGTH*)
- CONFIG_TIMER_TASK_PRIORITY (*CONFIG_FREERTOS_TIMER_TASK_PRIORITY*)
- CONFIG_TIMER_TASK_STACK_DEPTH (*CONFIG_FREERTOS_TIMER_TASK_STACK_DEPTH*)
- CONFIG_TIMER_TASK_STACK_SIZE (*CONFIG_ESP_TIMER_TASK_STACK_SIZE*)
- CONFIG_TOOLPREFIX (*CONFIG_SDK_TOOLPREFIX*)
- CONFIG_UDP_RECVMBOX_SIZE (*CONFIG_LWIP_UDP_RECVMBOX_SIZE*)
- CONFIG_ULP_COPROC_ENABLED (*CONFIG_ESP32_ULP_COPROC_ENABLED*)
- CONFIG_ULP_COPROC_RESERVE_MEM (*CONFIG_ESP32_ULP_COPROC_RESERVE_MEM*)
- CONFIG_USE_ONLY_LWIP_SELECT (*CONFIG_LWIP_USE_ONLY_LWIP_SELECT*)
- CONFIG_WARN_WRITE_STRINGS (*CONFIG_COMPILER_WARN_WRITE_STRINGS*)
- CONFIG_WIFI_LWIP_ALLOCATION_FROM_SPIRAM_FIRST (*CONFIG_SPIRAM_TRY_ALLOCATE_WIFI_LWIP*)

2.8.7 Customisations

Because IDF builds by default with *Warning On Undefined Variables*, when the Kconfig tool generates Makefiles (the `auto.conf` file) its behaviour has been customised. In normal Kconfig, a variable which is set to “no” is undefined. In IDF’s version of Kconfig, this variable is defined in the Makefile but has an empty value.

(Note that `ifdef` and `ifndef` can still be used in Makefiles, because they test if a variable is defined *and has a non-empty value.*)

When generating header files for C & C++, the behaviour is not customised - so `ifdef` can be used to test if a boolean config item is set or not.

2.9 Error Codes Reference

This section lists various error code constants defined in ESP-IDF.

For general information about error codes in ESP-IDF, see *Error Handling*.

ESP_FAIL (-1): Generic `esp_err_t` code indicating failure

ESP_OK (0): esp_err_t value indicating success (no error)

ESP_ERR_NO_MEM (0x101): Out of memory

ESP_ERR_INVALID_ARG (0x102): Invalid argument

ESP_ERR_INVALID_STATE (0x103): Invalid state

ESP_ERR_INVALID_SIZE (0x104): Invalid size

ESP_ERR_NOT_FOUND (0x105): Requested resource not found

ESP_ERR_NOT_SUPPORTED (0x106): Operation or feature not supported

ESP_ERR_TIMEOUT (0x107): Operation timed out

ESP_ERR_INVALID_RESPONSE (0x108): Received response was invalid

ESP_ERR_INVALID_CRC (0x109): CRC or checksum was invalid

ESP_ERR_INVALID_VERSION (0x10a): Version was invalid

ESP_ERR_INVALID_MAC (0x10b): MAC address was invalid

ESP_ERR_NOT_FINISHED (0x201)

ESP_ERR_NVS_BASE (0x1100): Starting number of error codes

ESP_ERR_NVS_NOT_INITIALIZED (0x1101): The storage driver is not initialized

ESP_ERR_NVS_NOT_FOUND (0x1102): Id namespace doesn't exist yet and mode is NVS_READONLY

ESP_ERR_NVS_TYPE_MISMATCH (0x1103): The type of set or get operation doesn't match the type of value stored in NVS

ESP_ERR_NVS_READ_ONLY (0x1104): Storage handle was opened as read only

ESP_ERR_NVS_NOT_ENOUGH_SPACE (0x1105): There is not enough space in the underlying storage to save the value

ESP_ERR_NVS_INVALID_NAME (0x1106): Namespace name doesn't satisfy constraints

ESP_ERR_NVS_INVALID_HANDLE (0x1107): Handle has been closed or is NULL

ESP_ERR_NVS_REMOVE_FAILED (0x1108): The value wasn't updated because flash write operation has failed. The value was written however, and update will be finished after re-initialization of nvs, provided that flash operation doesn't fail again.

ESP_ERR_NVS_KEY_TOO_LONG (0x1109): Key name is too long

ESP_ERR_NVS_PAGE_FULL (0x110a): Internal error; never returned by nvs API functions

ESP_ERR_NVS_INVALID_STATE (0x110b): NVS is in an inconsistent state due to a previous error. Call nvs_flash_init and nvs_open again, then retry.

ESP_ERR_NVS_INVALID_LENGTH (0x110c): String or blob length is not sufficient to store data

ESP_ERR_NVS_NO_FREE_PAGES (0x110d): NVS partition doesn't contain any empty pages. This may happen if NVS partition was truncated. Erase the whole partition and call nvs_flash_init again.

ESP_ERR_NVS_VALUE_TOO_LONG (0x110e): String or blob length is longer than supported by the implementation

ESP_ERR_NVS_PART_NOT_FOUND (0x110f): Partition with specified name is not found in the partition table

ESP_ERR_NVS_NEW_VERSION_FOUND (0x1110): NVS partition contains data in new format and cannot be recognized by this version of code

ESP_ERR_NVS_XTS_ENCR_FAILED (0x1111): XTS encryption failed while writing NVS entry

ESP_ERR_NVS_XTS_DECR_FAILED (0x1112): XTS decryption failed while reading NVS entry

ESP_ERR_NVS_XTS_CFG_FAILED (0x1113): XTS configuration setting failed

ESP_ERR_NVS_XTS_CFG_NOT_FOUND (0x1114): XTS configuration not found

ESP_ERR_NVS_ENCR_NOT_SUPPORTED (0x1115): NVS encryption is not supported in this version

ESP_ERR_NVS_KEYS_NOT_INITIALIZED (0x1116): NVS key partition is uninitialized

ESP_ERR_NVS_CORRUPT_KEY_PART (0x1117): NVS key partition is corrupt

ESP_ERR_NVS_CONTENT_DIFFERS (0x1118): Internal error; never returned by nvs API functions. NVS key is different in comparison

ESP_ERR_ULP_BASE (0x1200): Offset for ULP-related error codes

ESP_ERR_ULP_SIZE_TOO_BIG (0x1201): Program doesn't fit into RTC memory reserved for the ULP

ESP_ERR_ULP_INVALID_LOAD_ADDR (0x1202): Load address is outside of RTC memory reserved for the ULP

ESP_ERR_ULP_DUPLICATE_LABEL (0x1203): More than one label with the same number was defined

ESP_ERR_ULP_UNDEFINED_LABEL (0x1204): Branch instructions references an undefined label

ESP_ERR_ULP_BRANCH_OUT_OF_RANGE (0x1205): Branch target is out of range of B instruction (try replacing with BX)

ESP_ERR_OTA_BASE (0x1500): Base error code for ota_ops api

ESP_ERR_OTA_PARTITION_CONFLICT (0x1501): Error if request was to write or erase the current running partition

ESP_ERR_OTA_SELECT_INFO_INVALID (0x1502): Error if OTA data partition contains invalid content

ESP_ERR_OTA_VALIDATE_FAILED (0x1503): Error if OTA app image is invalid

ESP_ERR_OTA_SMALL_SEC_VER (0x1504): Error if the firmware has a secure version less than the running firmware.

ESP_ERR_OTA_ROLLBACK_FAILED (0x1505): Error if flash does not have valid firmware in passive partition and hence rollback is not possible

ESP_ERR_OTA_ROLLBACK_INVALID_STATE (0x1506): Error if current active firmware is still marked in pending validation state (ESP_OTA_IMG_PENDING_VERIFY), essentially first boot of firmware image post upgrade and hence firmware upgrade is not possible

ESP_ERR_EFUSE (0x1600): Base error code for efuse api.

ESP_OK_EFUSE_CNT (0x1601): OK the required number of bits is set.

ESP_ERR_EFUSE_CNT_IS_FULL (0x1602): Error field is full.

ESP_ERR_EFUSE_REPEATED_PROG (0x1603): Error repeated programming of programmed bits is strictly forbidden.

ESP_ERR_CODING (0x1604): Error while a encoding operation.

ESP_ERR_IMAGE_BASE (0x2000)

ESP_ERR_IMAGE_FLASH_FAIL (0x2001)

ESP_ERR_IMAGE_INVALID (0x2002)

ESP_ERR_WIFI_BASE (0x3000): Starting number of WiFi error codes

ESP_ERR_WIFI_NOT_INIT (0x3001): WiFi driver was not installed by esp_wifi_init

ESP_ERR_WIFI_NOT_STARTED (0x3002): WiFi driver was not started by esp_wifi_start

ESP_ERR_WIFI_NOT_STOPPED (0x3003): WiFi driver was not stopped by esp_wifi_stop

ESP_ERR_WIFI_IF (0x3004): WiFi interface error

ESP_ERR_WIFI_MODE (0x3005): WiFi mode error

ESP_ERR_WIFI_STATE (0x3006): WiFi internal state error

ESP_ERR_WIFI_CONN (0x3007): WiFi internal control block of station or soft-AP error

ESP_ERR_WIFI_NVS (0x3008): WiFi internal NVS module error

ESP_ERR_WIFI_MAC (0x3009): MAC address is invalid

ESP_ERR_WIFI_SSID (0x300a): SSID is invalid

ESP_ERR_WIFI_PASSWORD (0x300b): Password is invalid

ESP_ERR_WIFI_TIMEOUT (0x300c): Timeout error

ESP_ERR_WIFI_WAKE_FAIL (0x300d): WiFi is in sleep state(RF closed) and wakeup fail

ESP_ERR_WIFI_WOULD_BLOCK (0x300e): The caller would block

ESP_ERR_WIFI_NOT_CONNECT (0x300f): Station still in disconnect status

ESP_ERR_WIFI_POST (0x3012): Failed to post the event to WiFi task

ESP_ERR_WIFI_INIT_STATE (0x3013): Invalid WiFi state when init/deinit is called

ESP_ERR_WIFI_STOP_STATE (0x3014): Returned when WiFi is stopping

ESP_ERR_WIFI_REGISTRAR (0x3033): WPS registrar is not supported

ESP_ERR_WIFI_WPS_TYPE (0x3034): WPS type error

ESP_ERR_WIFI_WPS_SM (0x3035): WPS state machine is not initialized

ESP_ERR_ESPNOW_BASE (0x3064): ESPNOW error number base.

ESP_ERR_ESPNOW_NOT_INIT (0x3065): ESPNOW is not initialized.

ESP_ERR_ESPNOW_ARG (0x3066): Invalid argument

ESP_ERR_ESPNOW_NO_MEM (0x3067): Out of memory

ESP_ERR_ESPNOW_FULL (0x3068): ESPNOW peer list is full

ESP_ERR_ESPNOW_NOT_FOUND (0x3069): ESPNOW peer is not found

ESP_ERR_ESPNOW_INTERNAL (0x306a): Internal error

ESP_ERR_ESPNOW_EXIST (0x306b): ESPNOW peer has existed

ESP_ERR_ESPNOW_IF (0x306c): Interface error

ESP_ERR_MESH_BASE (0x4000): Starting number of MESH error codes

ESP_ERR_MESH_WIFI_NOT_START (0x4001)

ESP_ERR_MESH_NOT_INIT (0x4002)

ESP_ERR_MESH_NOT_CONFIG (0x4003)

ESP_ERR_MESH_NOT_START (0x4004)

ESP_ERR_MESH_NOT_SUPPORT (0x4005)

ESP_ERR_MESH_NOT_ALLOWED (0x4006)

ESP_ERR_MESH_NO_MEMORY (0x4007)

ESP_ERR_MESH_ARGUMENT (0x4008)

ESP_ERR_MESH_EXCEED_MTU (0x4009)

ESP_ERR_MESH_TIMEOUT (0x400a)

ESP_ERR_MESH_DISCONNECTED (0x400b)

ESP_ERR_MESH_QUEUE_FAIL (0x400c)

ESP_ERR_MESH_QUEUE_FULL (0x400d)

ESP_ERR_MESH_NO_PARENT_FOUND (0x400e)

ESP_ERR_MESH_NO_ROUTE_FOUND (0x400f)

ESP_ERR_MESH_OPTION_NULL (**0x4010**)

ESP_ERR_MESH_OPTION_UNKNOWN (**0x4011**)

ESP_ERR_MESH_XON_NO_WINDOW (**0x4012**)

ESP_ERR_MESH_INTERFACE (**0x4013**)

ESP_ERR_MESH_DISCARD_DUPLICATE (**0x4014**)

ESP_ERR_MESH_DISCARD (**0x4015**)

ESP_ERR_MESH_VOTING (**0x4016**)

ESP_ERR_MESH_XMIT (**0x4017**)

ESP_ERR_MESH_QUEUE_READ (**0x4018**)

ESP_ERR_MESH_RECV_RELEASE (**0x401a**)

ESP_ERR_ESP_NETIF_BASE (**0x5000**)

ESP_ERR_ESP_NETIF_INVALID_PARAMS (**0x5001**)

ESP_ERR_ESP_NETIF_IF_NOT_READY (**0x5002**)

ESP_ERR_ESP_NETIF_DHCP_START_FAILED (**0x5003**)

ESP_ERR_ESP_NETIF_DHCP_ALREADY_STARTED (**0x5004**)

ESP_ERR_ESP_NETIF_DHCP_ALREADY_STOPPED (**0x5005**)

ESP_ERR_ESP_NETIF_NO_MEM (**0x5006**)

ESP_ERR_ESP_NETIF_DHCP_NOT_STOPPED (**0x5007**)

ESP_ERR_ESP_NETIF_DRIVER_ATTACH_FAILED (**0x5008**)

ESP_ERR_ESP_NETIF_INIT_FAILED (**0x5009**)

ESP_ERR_ESP_NETIF_DNS_NOT_CONFIGURED (**0x500a**)

ESP_ERR_FLASH_BASE (**0x6000**): Starting number of flash error codes

ESP_ERR_FLASH_OP_FAIL (**0x6001**)

ESP_ERR_FLASH_OP_TIMEOUT (**0x6002**)

ESP_ERR_FLASH_NOT_INITIALISED (**0x6003**)

ESP_ERR_FLASH_UNSUPPORTED_HOST (**0x6004**)

ESP_ERR_FLASH_UNSUPPORTED_CHIP (**0x6005**)

ESP_ERR_FLASH_PROTECTED (**0x6006**)

ESP_ERR_HTTP_BASE (**0x7000**): Starting number of HTTP error codes

ESP_ERR_HTTP_MAX_REDIRECT (**0x7001**): The error exceeds the number of HTTP redirects

ESP_ERR_HTTP_CONNECT (**0x7002**): Error open the HTTP connection

ESP_ERR_HTTP_WRITE_DATA (**0x7003**): Error write HTTP data

ESP_ERR_HTTP_FETCH_HEADER (**0x7004**): Error read HTTP header from server

ESP_ERR_HTTP_INVALID_TRANSPORT (**0x7005**): There are no transport support for the input scheme

ESP_ERR_HTTP_CONNECTING (**0x7006**): HTTP connection hasn't been established yet

ESP_ERR_HTTP_EAGAIN (**0x7007**): Mapping of errno EAGAIN to esp_err_t

ESP_ERR_ESP_TLS_BASE (**0x8000**): Starting number of ESP-TLS error codes

ESP_ERR_ESP_TLS_CANNOT_RESOLVE_HOSTNAME (**0x8001**): Error if hostname couldn't be resolved upon tls connection

ESP_ERR_ESP_TLS_CANNOT_CREATE_SOCKET (**0x8002**): Failed to create socket

ESP_ERR_ESP_TLS_UNSUPPORTED_PROTOCOL_FAMILY (**0x8003**): Unsupported protocol family

ESP_ERR_ESP_TLS_FAILED_CONNECT_TO_HOST (**0x8004**): Failed to connect to host

ESP_ERR_ESP_TLS_SOCKET_SETOPT_FAILED (**0x8005**): failed to set socket option

ESP_ERR_MBEDTLS_CERT_PARTLY_OK (**0x8006**): mbedtls parse certificates was partly successful

ESP_ERR_MBEDTLS_CTR_DRBG_SEED_FAILED (**0x8007**): mbedtls api returned error

ESP_ERR_MBEDTLS_SSL_SET_HOSTNAME_FAILED (**0x8008**): mbedtls api returned error

ESP_ERR_MBEDTLS_SSL_CONFIG_DEFAULTS_FAILED (**0x8009**): mbedtls api returned error

ESP_ERR_MBEDTLS_SSL_CONF_ALPN_PROTOCOLS_FAILED (**0x800a**): mbedtls api returned error

ESP_ERR_MBEDTLS_X509_CERT_PARSE_FAILED (**0x800b**): mbedtls api returned error

ESP_ERR_MBEDTLS_SSL_CONF_OWN_CERT_FAILED (**0x800c**): mbedtls api returned error

ESP_ERR_MBEDTLS_SSL_SETUP_FAILED (**0x800d**): mbedtls api returned error

ESP_ERR_MBEDTLS_SSL_WRITE_FAILED (**0x800e**): mbedtls api returned error

ESP_ERR_MBEDTLS_PK_PARSE_KEY_FAILED (**0x800f**): mbedtls api returned failed

ESP_ERR_MBEDTLS_SSL_HANDSHAKE_FAILED (**0x8010**): mbedtls api returned failed

ESP_ERR_MBEDTLS_SSL_CONF_PSK_FAILED (**0x8011**): mbedtls api returned failed

ESP_ERR_ESP_TLS_CONNECTION_TIMEOUT (**0x8012**): new connection in esp_tls_low_level_conn connection timed out

ESP_ERR_HTTPS_OTA_BASE (**0x9000**)

ESP_ERR_HTTPS_OTA_IN_PROGRESS (**0x9001**)

ESP_ERR_PING_BASE (**0xa000**)

ESP_ERR_PING_INVALID_PARAMS (**0xa001**)

ESP_ERR_PING_NO_MEM (**0xa002**)

ESP_ERR_HTTPD_BASE (**0xb000**): Starting number of HTTPD error codes

ESP_ERR_HTTPD_HANDLERS_FULL (**0xb001**): All slots for registering URI handlers have been consumed

ESP_ERR_HTTPD_HANDLER_EXISTS (**0xb002**): URI handler with same method and target URI already registered

ESP_ERR_HTTPD_INVALID_REQ (**0xb003**): Invalid request pointer

ESP_ERR_HTTPD_RESULT_TRUNC (**0xb004**): Result string truncated

ESP_ERR_HTTPD_RESP_HDR (**0xb005**): Response header field larger than supported

ESP_ERR_HTTPD_RESP_SEND (**0xb006**): Error occurred while sending response packet

ESP_ERR_HTTPD_ALLOC_MEM (**0xb007**): Failed to dynamically allocate memory for resource

ESP_ERR_HTTPD_TASK (**0xb008**): Failed to launch server task/thread

Chapter 3

ESP32 Hardware Reference

3.1 ESP32 Modules and Boards

Espressif designs and manufactures different modules and development boards to help users evaluate the potential of the ESP32 family of chips.

This document provides description of modules and development boards currently available from Espressif.

Note: For description of previous versions of modules and development boards as well as for description of discontinued ones, please go to Section [Previous Versions of ESP32 Modules and Boards](#).

3.1.1 Modules

This is a family of ESP32-based modules with some integrated key components, including a crystal oscillator and an antenna matching circuit. The modules constitute ready-made solutions for integration into final products. If combined with a few extra components, such as a programming interface, bootstrapping resistors, and pin headers, these modules can also be used for evaluation of ESP32's functionality.

The key characteristics of these modules are summarized in the table below. Some additional details are covered in the following sections.

| Module | Chip | Flash, MB | PSRAM, MB | Ant. | Dimensions, mm |
|---------------------|--------------|-------------|-----------|------|-----------------|
| ESP32-WROOM-32 | ESP32-D0WDQ6 | 4 | – | MIFA | 18 × 25.5 × 3.1 |
| ESP32-WROOM-32D | ESP32-D0WD | 4, 8, or 16 | – | MIFA | 18 × 25.5 × 3.1 |
| ESP32-WROOM-32U | ESP32-D0WD | 4, 8, or 16 | – | U.FL | 18 × 19.2 × 3.1 |
| ESP32-SOLO-1 | ESP32-S0WD | 4 | – | MIFA | 18 × 25.5 × 3.1 |
| ESP32-WROVER (PCB) | ESP32-D0WDQ6 | 4 | 8 | MIFA | 18 × 31.4 × 3.3 |
| ESP32-WROVER (IPEX) | ESP32-D0WDQ6 | 4 | 8 | U.FL | 18 × 31.4 × 3.3 |
| ESP32-WROVER-B | ESP32-D0WD | 4, 8, or 16 | 8 | MIFA | 18 × 31.4 × 3.3 |
| ESP32-WROVER-IB | ESP32-D0WD | 4, 8, or 16 | 8 | U.FL | 18 × 31.4 × 3.3 |

- ESP32-**D**.. identifies a dual-core chip, ESP32-**S**.. identifies a single-core chip
- MIFA - Meandered Inverted-F Antenna
- U.FL - U.FL / IPEX antenna connector
- ESP32-WROOM-32x, ESP32-WROVER-B and ESP32-WROVER-IB modules come with 4 MB flash by default but also available with custom flash sizes of 8 MB and 16 MB, see [Espressif Products Ordering Information](#) (PDF)
- [ESP32 Chip Datasheet](#) (PDF)
- Initial release of the ESP32-WROVER module had 4 MB of PSRAM
- *ESP32-WROOM-32* was previously called *ESP-WROOM-32*

ESP32-WROOM-32

This is a basic and commonly adopted ESP32 module with the ESP32-D0WDQ6 chip on board. It was the first module of the WROOM / WROVER family released to the market.

For key characteristics, see the table in Section [Modules](#), [Espressif Products Ordering Information](#).

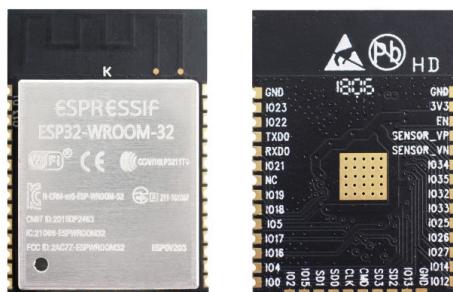


Fig. 1: ESP32-WROOM-32 module (front and back)

Documentation

- [ESP32-WROOM-32 Datasheet \(PDF\)](#)
- [ESP32-WROOM-32 Reference Design](#) containing OrCAD schematic, PCB layout, gerber and BOM files

ESP32-WROOM-32D / ESP32-WROOM-32U

Both modules integrate the ESP32-D0WD chip which has a smaller footprint than the chip ESP32-D0WDQ6 installed in [ESP32-WROOM-32](#).

For key characteristics, see the table in Section [Modules](#) and [Espressif Products Ordering Information](#).

ESP32-WROOM-32U is the smallest representative of the whole WROOM / WROVER family of modules.

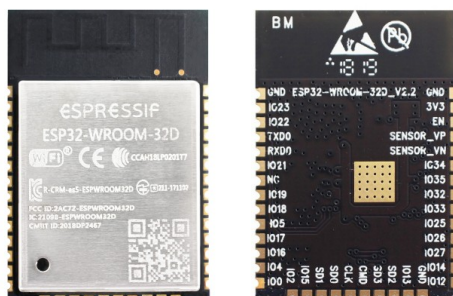


Fig. 2: ESP32-WROOM-32D module (front and back)

Documentation

- [ESP32-WROOM-32D / ESP32-WROOM-32U Datasheet \(PDF\)](#)

ESP32-SOLO-1

This is a simplified version of the ESP32-WROOM-32D module. It contains a single-core ESP32 chip that supports a clock frequency of up to 160 MHz.

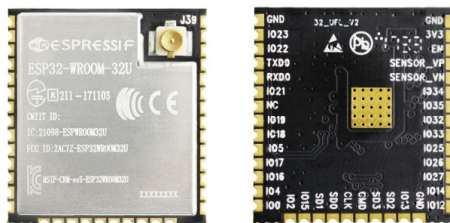


Fig. 3: ESP32-WROOM-32U module (front and back)

For key characteristics, see the table in Section [Modules](#) and [Espressif Products Ordering Information](#).

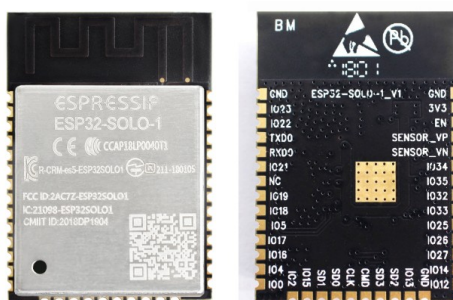


Fig. 4: ESP32-SOLO-1 module (front and back)

Documentation

- [ESP32-SOLO-1 Datasheet \(PDF\)](#)

ESP32-WROVER series

This series consists of a few modifications of ESP32-WROOM-32x modules, which among other upgrades include additional 8 MB SPI PSRAM (pseudo static RAM).

For details, see the table in Section [Modules](#) and [Espressif Products Ordering Information](#).

- **ESP32-WROVER (PCB)** and **ESP32-WROVER (IPEX)** have PSRAM that operates at 1.8 V and supports up to 144 MHz clock rate.
- **ESP32-WROVER-B** and **ESP32-WROVER-IB** have PSRAM that operates at 3.3 V and supports up to 133 MHz clock rate.

The picture below shows an ESP32-WROVER module with a PCB antenna.

Documentation

- [ESP32-WROVER Datasheet \(PDF\)](#)
- [ESP32-WROVER-B Datasheet \(PDF\)](#)
- [ESP-PSRAM64 & ESP-PSRAM64H Datasheet \(PDF\)](#)
- [ESP32-WROVER Reference Design](#) containing OrCAD schematic, PCB layout, gerber and BOM files

ESP32-PICO-D4

ESP32-PICO-D4 is a System-in-Package (SiP) module, integrating all peripheral components seamlessly, including the following:

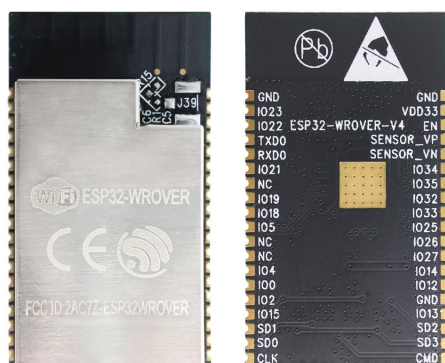


Fig. 5: ESP32-WROVER module (front and back)

- 4 MB flash memory
- crystal oscillator
- filter capacitors
- RF matching circuit

For key characteristics, see [Espressif Products Ordering Information](#).

Documentation

- [ESP32-PICO-D4 Datasheet \(PDF\)](#)

3.1.2 Development Boards

Depending on the intended functionality, different development boards feature:

- Access to different ESP32 GPIO pins.
- Different interfaces: USB, JTAG.
- Different peripherals: touchpads, LCD screens, SD card slots, female headers for camera modules, etc.

ESP32-PICO-KIT V4.1

This is the smallest available ESP32-based development board. It features all the components for direct connection to a computer's USB port as well as pin headers for plugging into a mini breadboard.

The board is equipped with the [ESP32-PICO-D4](#) module. With such a module, the creation of a fully functional development board required only a few external components that fit on a PCB as small as 20 x 52 mm. The external components include antenna, LDO, USB-UART bridge, and two buttons for reset and activation of Firmware Download mode.

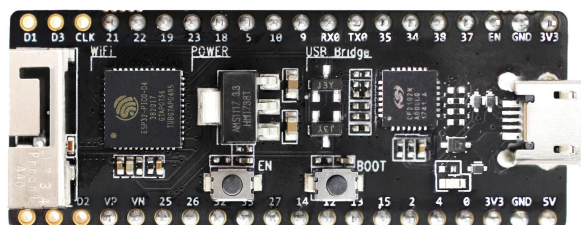


Fig. 6: ESP32-PICO-KIT V4.1 board

Comparing to ESP32-PICO-KIT V4, this version features the CP2102N USB-UART bridge that provides faster transfer rates of up to 3 Mbps.

Documentation

- [ESP32-PICO-KIT V4 / V4.1 Getting Started Guide](#)
- [ESP32-PICO-KIT V4.1 Schematic \(PDF\)](#)
- [ESP32-PICO-KIT Reference Design](#) containing OrCAD schematic, PCB layout, gerber and BOM files
- [ESP32-PICO-D4 Datasheet \(PDF\)](#)

Previous Versions

- [ESP32-PICO-KIT V4](#)
- [ESP32-PICO-KIT V3](#)

ESP32 DevKitC V4

This is a small and convenient development board that features:

- [ESP32-WROOM-32](#) module
- USB-to-serial programming interface that also provides power supply for the board
- pin headers
- pushbuttons for reset and activation of Firmware Download mode
- a few other components

Comparing to the previous [ESP32 Core Board V2 / ESP32 DevKitC](#), this version can integrate [ESP32-WROVER series](#) module instead of ESP32-WROOM-32 and has the CP2102N chip that supports faster baud rates.

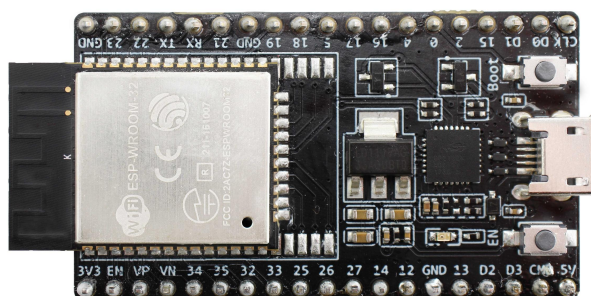


Fig. 7: ESP32 DevKitC V4 board

Documentation

- [ESP32-DevKitC V4 Getting Started Guide](#)
- [ESP32-DevKitC schematic \(PDF\)](#)
- [ESP32-DevKitC Reference Design](#) containing OrCAD schematic, PCB layout, gerber and BOM files
- [CP210x USB to UART Bridge VCP Drivers](#)

Previous Versions

- [ESP32 Core Board V2 / ESP32 DevKitC](#)

ESP-WROVER-KIT V4.1

This board features:

- Dual port USB-to-serial converter for programming
- JTAG interface for debugging
- MicroSD card slot
- 3.2" SPI LCD screen
- Female headers for a camera module

- RGB LED for diagnostics
- 32.768 kHz XTAL for internal RTC to operate it in low power modes

Power can be supplied either via USB or via a standard 5 mm power supply jack. A power source can be selected with a jumper and can be turned on/off with a separate switch.

This version of the ESP-WROVER-KIT board integrates the ESP-WROVER-B module that has 8 MB PSRAM for flexible extended storage and data processing capabilities. The board can accommodate other versions of ESP modules described in [Modules](#).

Comparing to [ESP-WROVER-KIT V3](#), this board has the following design changes:

- JP8, JP11, and JP13 have been combined into a single JP2.
- USB connector has been changed to DIP type and moved to the lower right corner of the board.
- R61 has been changed to a Zero-ohm resistor.
- Some components have been replaced with functional equivalents based on test results and sourcing options, e.g., the EN and Boot buttons.

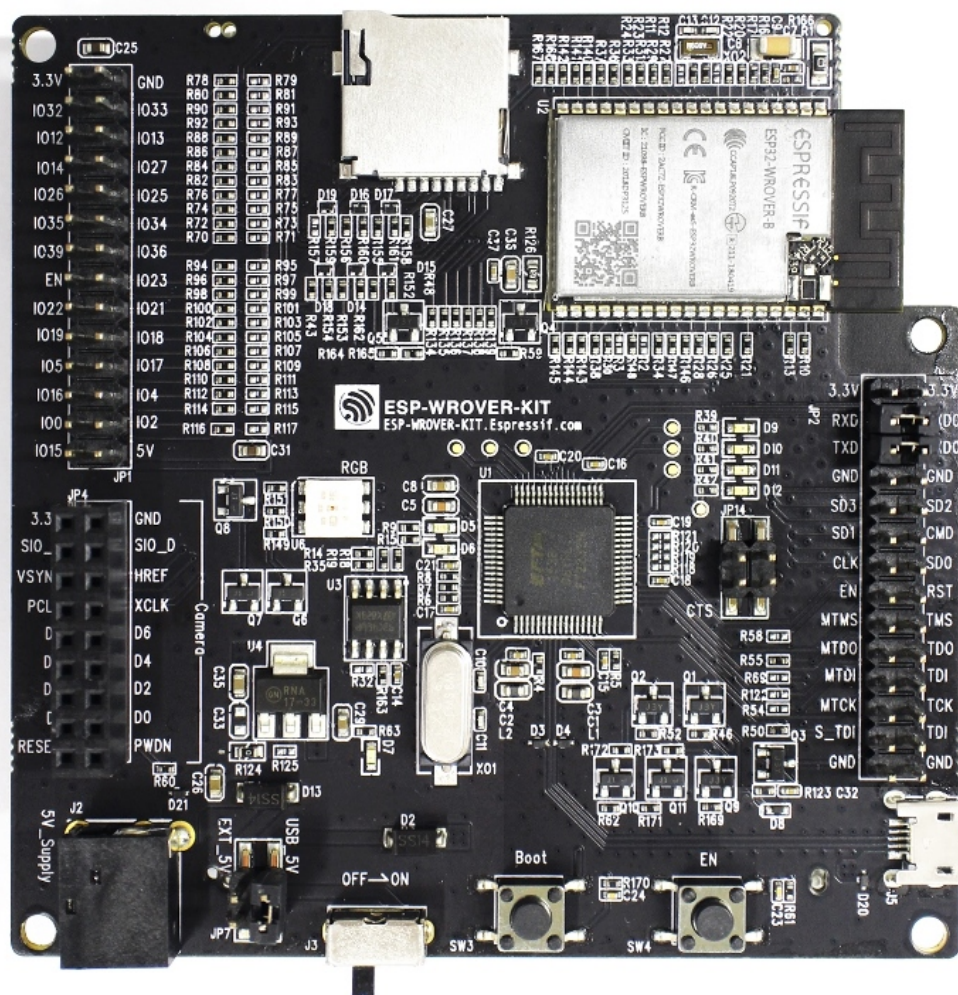


Fig. 8: ESP-WROVER-KIT V4.1 board

The board in the picture above integrates the ESP32-WROVER-B module.

Documentation

- [ESP-WROVER-KIT V4.1 Getting Started Guide](#)

- [ESP-WROVER-KIT V4.1 Schematic \(PDF\)](#)
- [JTAG Debugging](#)
- [FTDI Virtual COM Port Drivers](#)

Previous Versions

- [ESP-WROVER-KIT V3](#)
- [ESP-WROVER-KIT V2](#)
- [ESP-WROVER-KIT V1 / ESP32 DevKitJ V1](#)

3.1.3 Related Documents

- [Previous Versions of ESP32 Modules and Boards](#)

3.2 Previous Versions of ESP32 Modules and Boards

This section contains overview and links to documentation of previous version ESP32 Modules and Boards that have been replaced with newer versions or discontinued. It is maintained for convenience of users as previous versions of some modules and boards are still in use and some may still be available for purchase.

3.2.1 Modules (No updated or discontinued modules)

So far, no modules have been updated or discontinued.

3.2.2 Development Boards

To see the latest development boards, please refer to section [ESP32 Modules and Boards](#).

ESP32-PICO-KIT V4

The smallest ESP32 development board with all the components required to connect it directly to a PC USB port, and pin headers to plug into a mini breadboard. It is equipped with ESP32-PICO-D4 module that integrates 4 MB flash memory, a crystal oscillator, filter capacitors and RF matching circuit in one single package. As result, the fully functional development board requires only a few external components that can easily fit on a 20 x 52 mm PCB including antenna, LDO, USB-UART bridge and two buttons to reset it and put into download mode.

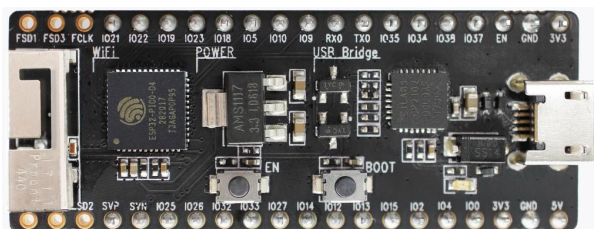


Fig. 9: ESP32-PICO-KIT V4 board

Comparing to ESP32-PICO-KIT V3, this version has revised printout and reduced number of exposed pins. Instead of 20, only 17 header pins are populated, so V4 can fit into a mini breadboard.

Documentation

- [ESP32-PICO-KIT V4 / V4.1 Getting Started Guide](#)
- [ESP32-PICO-KIT V4 Schematic \(PDF\)](#)
- [ESP32-PICO-D4 Datasheet \(PDF\)](#)

ESP32-PICO-KIT V3

The first public release of Espressif's ESP32-PICO-D4 module on a mini development board. The board has a USB port for programming and debugging and two rows of 20 pin headers to plug into a breadboard. The ESP32-PICO-D4 module itself is small and requires only a few external components. Besides two core CPUs it integrates 4MB flash memory, a crystal oscillator and antenna matching components in one single 7 x 7 mm package. As a result the module and all the components making the complete development board fit into 20 x 52 mm PCB.

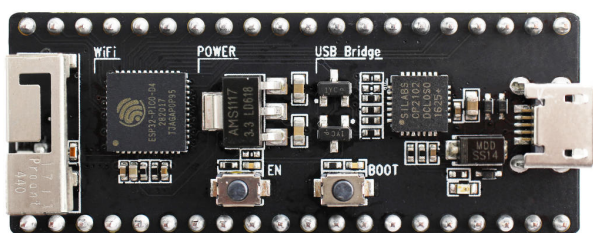


Fig. 10: ESP32-PICO-KIT V3 board

Documentation

- [ESP32-PICO-KIT V3 Getting Started Guide](#)
- [ESP32-PICO-KIT V3 Schematic \(PDF\)](#)
- [ESP32-PICO-D4 Datasheet \(PDF\)](#)

ESP32 Core Board V2 / ESP32 DevKitC

Small and convenient development board with ESP-WROOM-32 module installed, break out pin headers and minimum additional components. Includes USB to serial programming interface, that also provides power supply for the board. Has pushbuttons to reset the board and put it in upload mode.

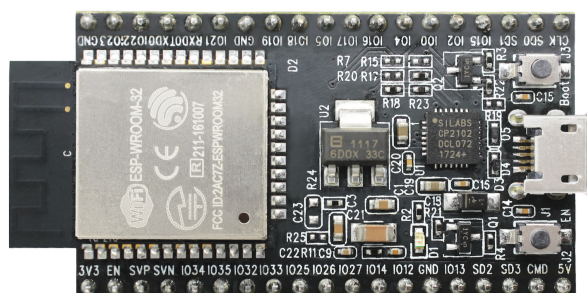


Fig. 11: ESP32 Core Board V2 / ESP32 DevKitC board

Documentation

- [ESP32-DevKitC V2 Getting Started Guide](#)
- [ESP32 DevKitC V2 Schematic \(PDF\)](#)
- [CP210x USB to UART Bridge VCP Drivers](#)

ESP-WROVER-KIT V3

The ESP-WROVER-KIT V3 development board has dual port USB to serial converter for programming and JTAG interface for debugging. Power supply is provided by USB interface or from standard 5 mm power supply jack. Power supply selection is done with a jumper and may be put on/off with a separate switch. This board has MicroSD card slot, 3.2" SPI LCD screen and dedicated header to connect a camera. It provides RGB diode for diagnostics. Includes 32.768 kHz XTAL for internal RTC to operate it in low power modes.

As all previous versions of ESP-WROVER-KIT boards, it is ready to accommodate an *ESP32-WROOM-32* or *ESP32-WROVER series* module.

This is the first release of ESP-WROVER-KIT shipped with *ESP32-WROVER series* module installed by default. This release also introduced several design changes to conditioning and interlocking of signals to the bootstrapping pins. Also, a zero Ohm resistor (R166) has been added between WROVER/WROOM module and VDD33 net, which can be desoldered, or replaced with a shunt resistor, for current measurement. This is intended to facilitate power consumption analysis in various operation modes of ESP32. Refer to schematic - the changes are enclosed in green border.

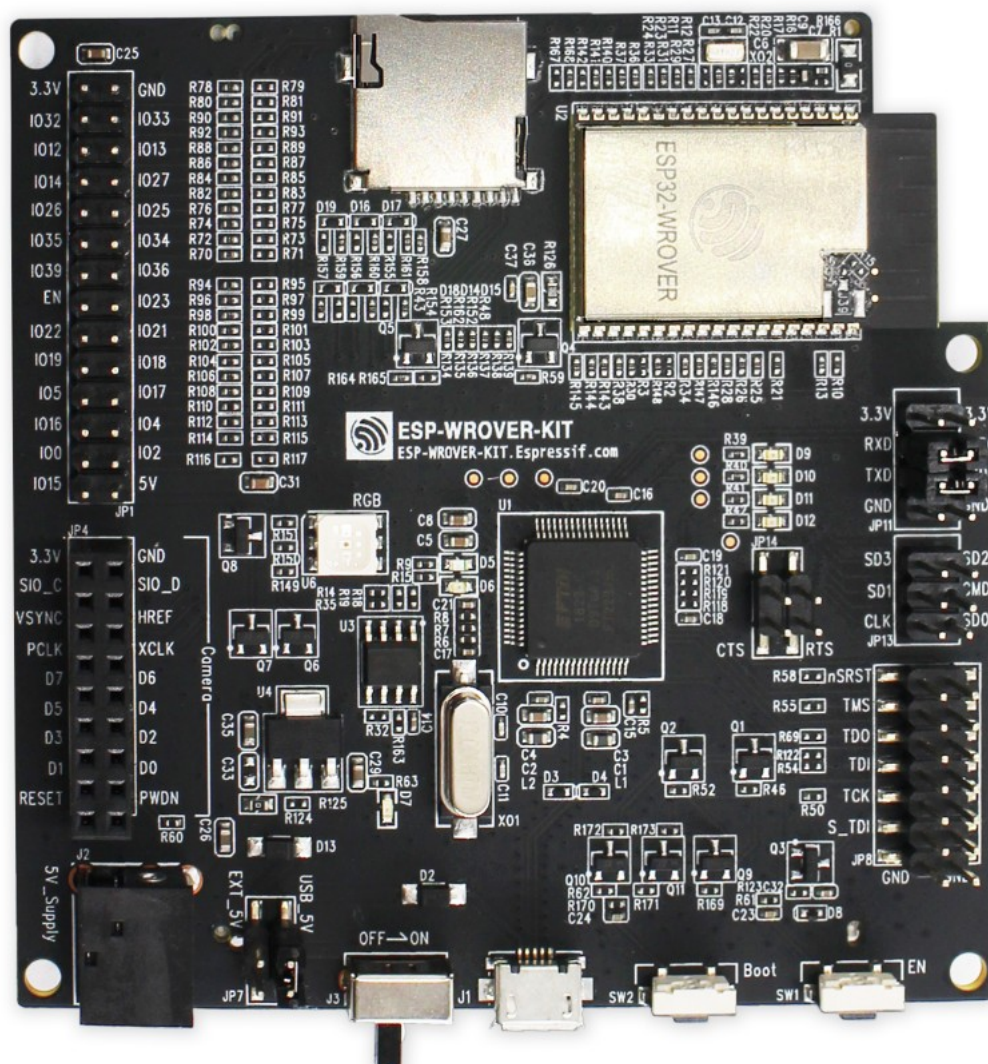


Fig. 12: ESP-WROVER-KIT V3 board

The camera header has been changed from male back to female. The board soldermask is matte black. The board on picture above has *ESP32-WROVER series* is installed.

Documentation

- [ESP-WROVER-KIT V3 Getting Started Guide](#)
- [ESP-WROVER-KIT V3 Schematic \(PDF\)](#)
- [JTAG Debugging](#)
- [FTDI Virtual COM Port Drivers](#)

ESP-WROVER-KIT V2

This is updated version of ESP32 DevKitJ V1 described above with design improvements identified when DevKitJ was in use, e.g. improved support for SD card. By default board has ESP-WROOM-32 module installed.

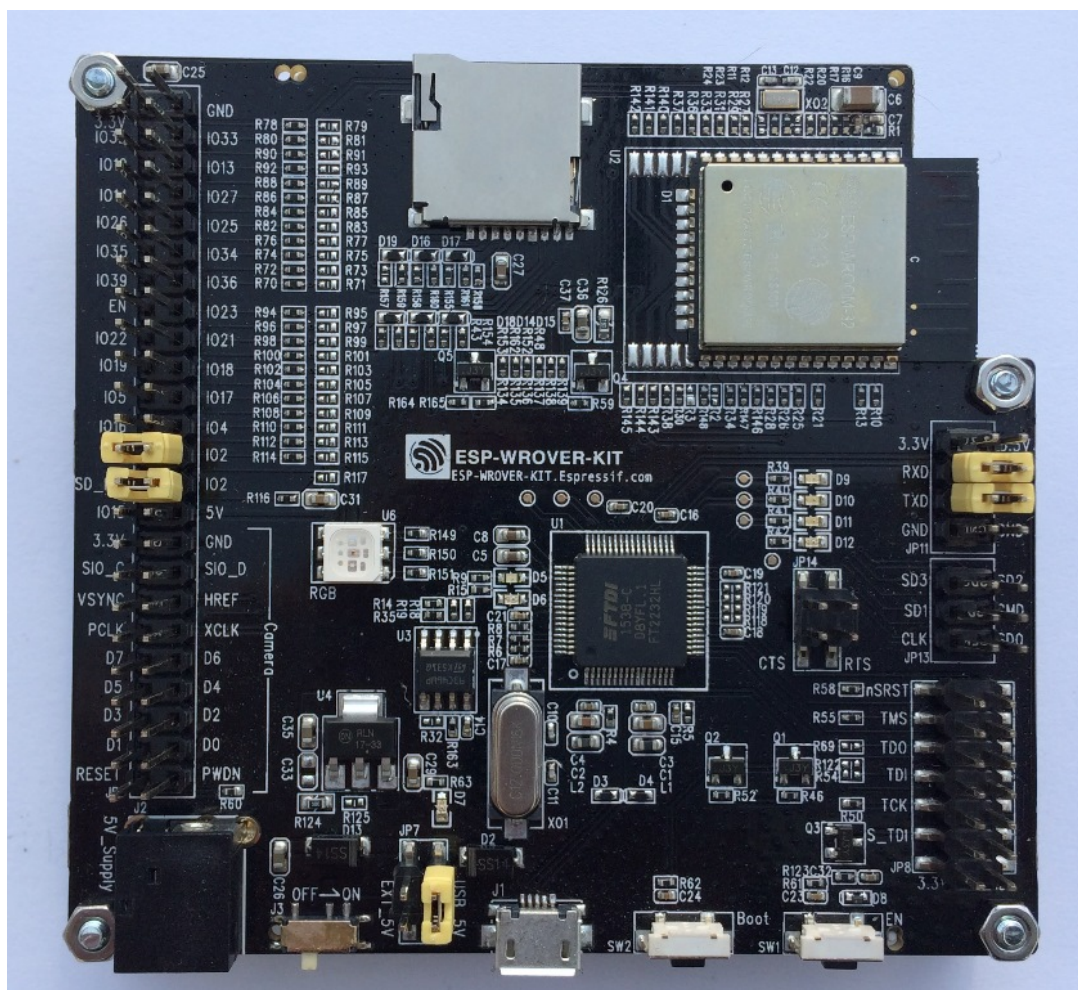


Fig. 13: ESP-WROVER-KIT V2 board

Comparing to previous version, this board has a shiny black finish and a male camera header.

Documentation

- [ESP-WROVER-KIT V2 Getting Started Guide](#)
- [ESP-WROVER-KIT V2 Schematic \(PDF\)](#)
- [JTAG Debugging](#)
- [FTDI Virtual COM Port Drivers](#)

ESP-WROVER-KIT V1 / ESP32 DevKitJ V1

The first version of ESP-WROVER-KIT development board. Shipped with ESP-WROOM-32 on board.

ESP-WROVER-KIT has dual port USB to serial converter for programming and JTAG interface for debugging. Power supply is provided by USB interface or from standard 5 mm power supply jack. Power supply selection is done with a jumper and may be put on/off with a separate switch. The board has MicroSD card slot, 3.2" SPI LCD screen and dedicated header to connect a camera. It provides RGB diode for diagnostics. Includes 32.768 kHz XTAL for internal RTC to operate it in low power modes.

All versions of ESP-WROVER-KIT are ready to accommodate an ESP-WROOM-32 or ESP32-WROVER module.

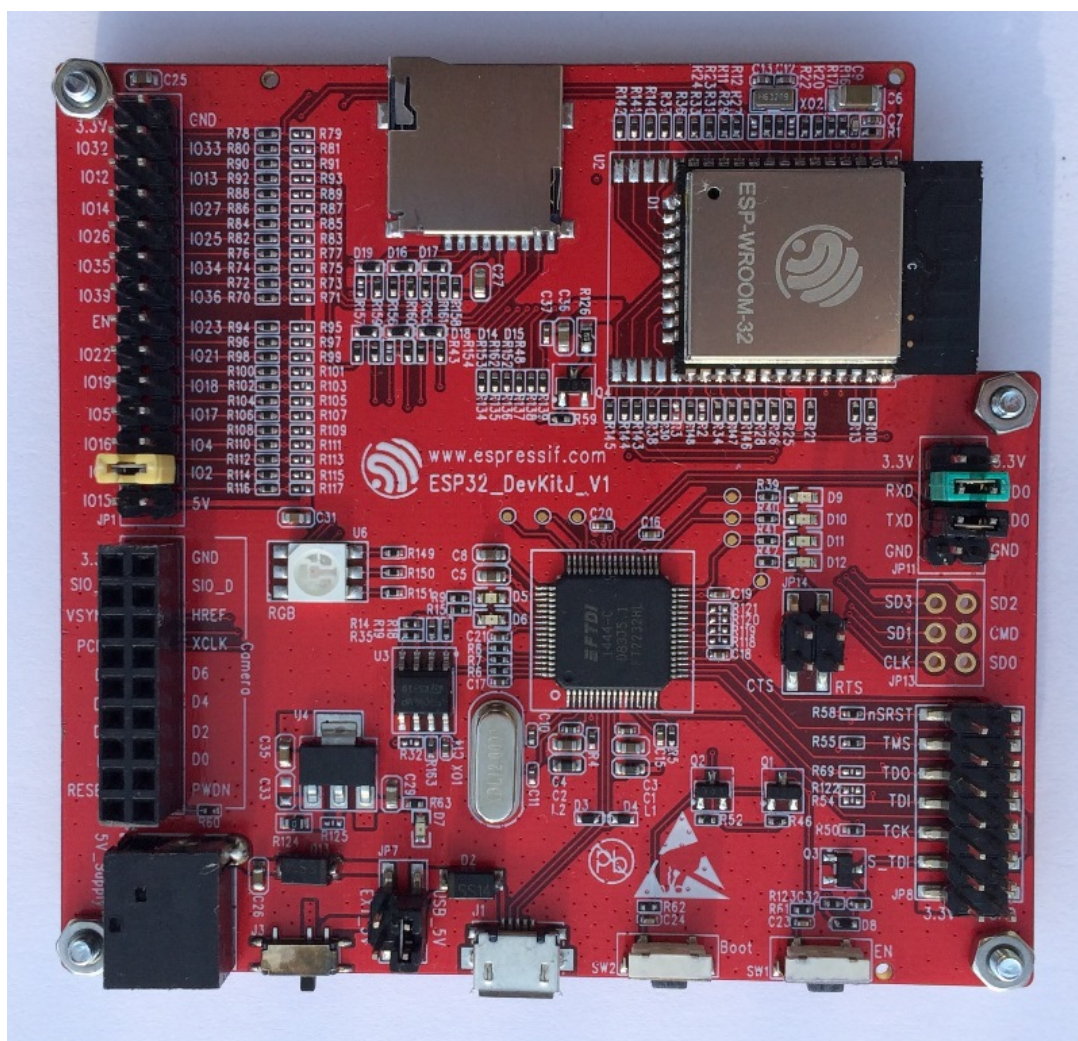


Fig. 14: ESP-WROVER-KIT V1 / ESP32 DevKitJ V1 board

The board has red soldermask.

Documentation

- [ESP-WROVER-KIT V1 Schematic \(PDF\)](#)
- [JTAG Debugging](#)
- [FTDI Virtual COM Port Drivers](#)

ESP32 Demo Board V2

One of first feature rich evaluation boards that contains several pin headers, dip switches, USB to serial programming interface, reset and boot mode press buttons, power switch, 10 touch pads and separate header to connect LCD screen.

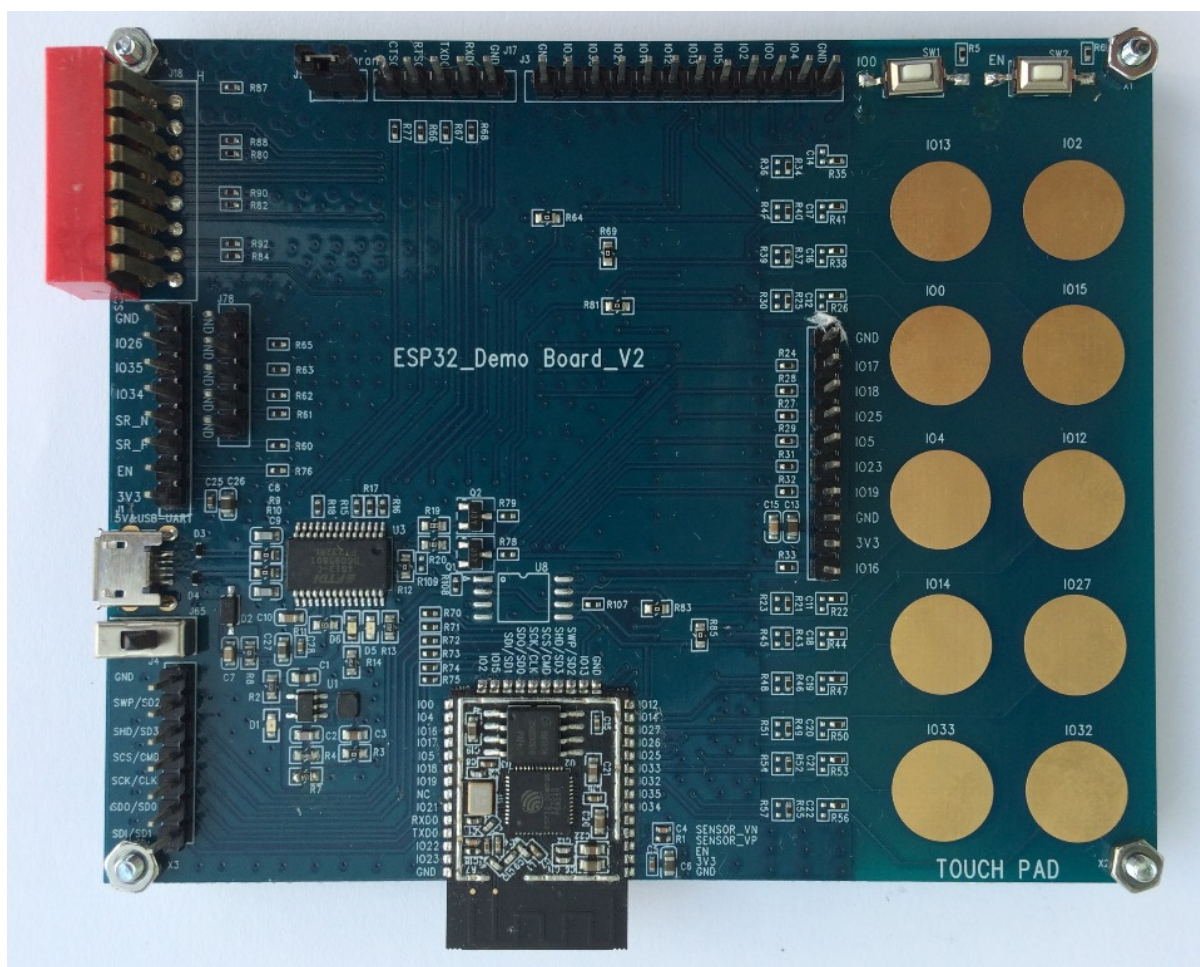


Fig. 15: ESP32 Demo Board V2

Production of this board is discontinued.

Documentation

- [ESP32 Demo Board V2 Schematic \(PDF\)](#)
- [FTDI Virtual COM Port Drivers](#)

3.2.3 Related Documents

- [ESP32 Modules and Boards](#)

Chapter 4

ESP32-S2 Beta Preview Support

The current master branch of ESP-IDF (pre-V4.1) contains preview support for ESP32-S2 Beta.

The ESP32-S2 Beta chip is not the same as the final ESP32-S2, the Beta chip is a limited engineering sample and as a result not all features are available.

Software support is still being developed ahead of the final ESP32-S2 release (see [Limitations](#) and [Roadmap](#) sections below).

4.1 Documentation

Except where it specifically mentions ESP32-S2, this version of the ESP-IDF Programming Guide is written for ESP32 only. Most software APIs and components are the same or very similar for ESP32-S2, but differences may not be documented yet.

4.2 Setting up for ESP32-S2 Beta

To start with, follow the [Get Started](#) guide to configure the CMake-based build system for ESP32.

When running `install.sh` or `install.bat` as part of the Get Started guide, the ESP32-S2 toolchain will be automatically downloaded and added to the IDF Tools directory. If you had already run `install.sh` or `install.bat` before updating to a version with ESP32-S2 support, you may need to re-run it.

The toolchain tuple is `xtensa-esp32s2-elf`, for example the GCC compiler for this target is named `xtensa-esp32s2-elf-gcc`.

4.2.1 Manual toolchain URLs

If not using `install.sh` or `install.bat`, you can download the manual toolchain manually from here:

- [Windows](#)
- [macOS](#)
- [Linux 64-bit x86](#)
- [Linux 32-bit x86](#)

If installing the toolchain manually, unpack it somewhere and add the `bin` subdirectory to your `PATH`.

4.3 Building a project for ESP32-S2 Beta

Important: Not all ESP-IDF examples support ESP32-S2 Beta.

For some examples this is because required hardware is not included in ESP32-S2 so it cannot be supported. For some examples this is because the example has not yet been updated for ESP32-S2.

If building an example, please check the example CMakeLists.txt file for the clause `SUPPORTED_TARGETS esp32`. If this is present, the example won't work on ESP32-S2.

Before building an ESP-IDF project, run the following command to switch the target to ESP32-S2 Beta:

```
idf.py set-target esp32s2beta
```

Important: Running `set-target` will clear the project configuration and create a new empty `sdkconfig` file for the project. The old configuration will be saved in `sdkconfig.old`.

Then configure the project:

```
idf.py menuconfig
```

Important: There are 3 versions of ESP32-S2 Beta engineering sample available with different Wi-Fi PHYs. They can be identified by the marking on the chip itself. The project must be configured to match the chip via the project setting “ESP32-S2 Beta chip version” (`CONFIG_ESP32S2_BETA_CHIP_VERSION`). This can be found under Component Config -> ESP32S2-specific. If the project setting doesn't match the chip then Wi-Fi performance may be very poor, or the chip may crash when initializing Wi-Fi.

Build and flash the project as usual, for example:

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

Consult the [Get Started](#) guide for more details on building and flashing an ESP-IDF project.

Note: It is possible to set the project target back to ESP32 by running `idf.py set-target esp32`. This also clears the project configuration.

4.4 Limitations

ESP32-S2 Beta support is currently a preview and does not support all features. Espressif firmware developers are working hard on the remaining support, so please update the `master` branch regularly to receive the latest code.

- The integrated USB OTG peripheral is not supported
- Documentation not updated for ESP32-S2 in most places
- Peripheral drivers are a work in progress
- No power saving modes
- No hardware security features or cryptographic accelerator support
- Time-of-Flight (TOF) measurements with normal Wi-Fi packets is not supported

4.5 Roadmap

- ESP-IDF V4.1 will support ESP32-S2 Beta chip as a preview with some limitations.

- ESP-IDF V4.2 will support ESP32-S2.

Support for ESP32-S2 Beta will be removed from ESP-IDF once ESP32-S2 is available.

4.6 Issues

If you find issues which are not mentioned in *Limitations*, please [open an issue on GitHub](#).

Please make sure to mention that you are using the ESP32-S2 Beta preview support and give your exact version of ESP-IDF.

Chapter 5

API Guides

5.1 Application Level Tracing library

5.1.1 Overview

IDF provides useful feature for program behavior analysis: application level tracing. It is implemented in the corresponding library and can be enabled in menuconfig. This feature allows to transfer arbitrary data between host and ESP32 via JTAG interface with small overhead on program execution.

Developers can use this library to send application specific state of execution to the host and receive commands or other type of info in the opposite direction at runtime. The main use cases of this library are:

1. Collecting application specific data, see [Application Specific Tracing](#)
2. Lightweight logging to the host, see [Logging to Host](#)
3. System behavior analysis, see [System Behavior Analysis with SEGGER SystemView](#)
4. Source code coverage, see [Gcov \(Source Code Coverage\)](#)

Tracing components when working over JTAG interface are shown in the figure below.

5.1.2 Modes of Operation

The library supports two modes of operation:

Post-mortem mode. This is the default mode. The mode does not need interaction with the host side. In this mode tracing module does not check whether host has read all the data from *HW UP BUFFER* buffer and overwrites old data with the new ones. This mode is useful when only the latest trace data are interesting to the user, e.g. for analyzing program's behavior just before the crash. Host can read the data later on upon user request, e.g. via special OpenOCD command in case of working via JTAG interface.

Streaming mode. Tracing module enters this mode when host connects to ESP32. In this mode before writing new data to *HW UP BUFFER* tracing module checks that there is enough space in it and if necessary waits for the host to read data and free enough memory. Maximum waiting time is controlled via timeout values passed by users to corresponding API routines. So when application tries to write data to trace buffer using finite value of the maximum waiting time it is possible situation that this data will be dropped. Especially this is true for tracing from time critical code (ISRs, OS scheduler code etc.) when infinite timeouts can lead to system malfunction. In order to avoid loss of such critical data developers can enable additional data buffering via menuconfig option [CONFIG_APPTRACE_PENDING_DATA_SIZE_MAX](#). This macro specifies the size of data which can be buffered in above conditions. The option can also help to overcome situation when data transfer to the host is temporarily slowed down, e.g. due to USB bus congestions etc. But it will not help when average bitrate of trace data stream exceeds HW interface capabilities.

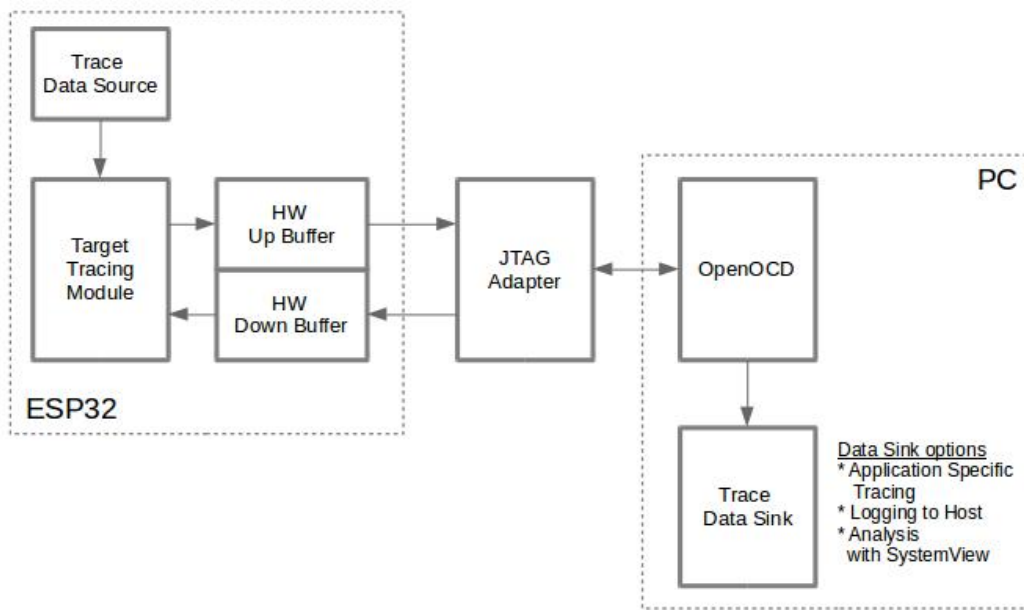


Fig. 1: Tracing Components when Working Over JTAG

5.1.3 Configuration Options and Dependencies

Using of this feature depends on two components:

1. **Host side:** Application tracing is done over JTAG, so it needs OpenOCD to be set up and running on host machine. For instructions on how to set it up, please see [JTAG Debugging](#) for details.
2. **Target side:** Application tracing functionality can be enabled in menuconfig. *Component config > Application Level Tracing* menu allows selecting destination for the trace data (HW interface for transport). Choosing any of the destinations automatically enables `CONFIG_APPTRACE_ENABLE` option.

Note: In order to achieve higher data rates and minimize number of dropped packets it is recommended to optimize setting of JTAG clock frequency, so it is at maximum and still provides stable operation of JTAG, see [Optimize JTAG speed](#).

There are two additional menuconfig options not mentioned above:

1. *Threshold for flushing last trace data to host on panic* (`CONFIG_APPTRACE_POSTMORTEM_FLUSH_THRESH`). This option is necessary due to the nature of working over JTAG. In that mode trace data are exposed to the host in 16 KB blocks. In post-mortem mode when one block is filled it is exposed to the host and the previous one becomes unavailable. In other words trace data are overwritten in 16 KB granularity. On panic the latest data from the current input block are exposed to host and host can read them for post-analysis. System panic may occur when very small amount of data are not exposed to the host yet. In this case the previous 16 KB of collected data will be lost and host will see the latest, but very small piece of the trace. It can be insufficient to diagnose the problem. This menuconfig option allows avoiding such situations. It controls the threshold for flushing data in case of panic. For example user can decide that it needs not less than 512 bytes of the recent trace data, so if there is less than 512 bytes of pending data at the moment of panic they will not be flushed and will not overwrite previous 16 KB. The option is only meaningful in post-mortem mode and when working over JTAG.
2. *Timeout for flushing last trace data to host on panic* (`CONFIG_APPTRACE_ONPANIC_HOST_FLUSH_TMO`). The option is only meaningful in streaming mode and controls the maximum time tracing module will wait for the host to read the last data in case of panic.

5.1.4 How to use this library

This library provides API for transferring arbitrary data between host and ESP32. When enabled in menuconfig target application tracing module is initialized automatically at the system startup, so all what the user needs to do is to call corresponding API to send, receive or flush the data.

Application Specific Tracing

In general user should decide what type of data should be transferred in every direction and how these data must be interpreted (processed). The following steps must be performed to transfer data between target and host:

1. On target side user should implement algorithms for writing trace data to the host. Piece of code below shows an example how to do this.

```
#include "esp_app_trace.h"
...
char buf[] = "Hello World!";
esp_err_t res = esp_apptrace_write(ESP_APPTRACE_DEST_TRAX, buf, strlen(buf),
↳ESP_APPTRACE_TMO_INFINITE);
if (res != ESP_OK) {
    ESP_LOGE(TAG, "Failed to write data to host!");
    return res;
}
```

`esp_apptrace_write()` function uses `memcpy` to copy user data to the internal buffer. In some cases it can be more optimal to use `esp_apptrace_buffer_get()` and `esp_apptrace_buffer_put()` functions. They allow developers to allocate buffer and fill it themselves. The following piece of code shows how to do this.

```
#include "esp_app_trace.h"
...
int number = 10;
char *ptr = (char *)esp_apptrace_buffer_get(ESP_APPTRACE_DEST_TRAX, 32, 100/
↳*tmo in us*);
if (ptr == NULL) {
    ESP_LOGE(TAG, "Failed to get buffer!");
    return ESP_FAIL;
}
sprintf(ptr, "Here is the number %d", number);
esp_err_t res = esp_apptrace_buffer_put(ESP_APPTRACE_DEST_TRAX, ptr, 100/*tmo_
↳in us*);
if (res != ESP_OK) {
    /* in case of error host tracing tool (e.g. OpenOCD) will report_
↳incomplete user buffer */
    ESP_LOGE(TAG, "Failed to put buffer!");
    return res;
}
```

Also according to his needs user may want to receive data from the host. Piece of code below shows an example how to do this.

```
#include "esp_app_trace.h"
...
char buf[32];
char down_buf[32];
size_t sz = sizeof(buf);

/* config down buffer */
esp_apptrace_down_buffer_config(down_buf, sizeof(down_buf));
/* check for incoming data and read them if any */
esp_err_t res = esp_apptrace_read(ESP_APPTRACE_DEST_TRAX, buf, &sz, 0/*do not_
↳wait*);
```

(continues on next page)

(continued from previous page)

```

if (res != ESP_OK) {
    ESP_LOGE(TAG, "Failed to read data from host!");
    return res;
}
if (sz > 0) {
    /* we have data, process them */
    ...
}

```

esp_appttrace_read() function uses memcpy to copy host data to user buffer. In some cases it can be more optimal to use esp_appttrace_down_buffer_get() and esp_appttrace_down_buffer_put() functions. They allow developers to occupy chunk of read buffer and process it in-place. The following piece of code shows how to do this.

```

#include "esp_app_trace.h"
...
char down_buf[32];
uint32_t *number;
size_t sz = 32;

/* config down buffer */
esp_appttrace_down_buffer_config(down_buf, sizeof(down_buf));
char *ptr = (char *)esp_appttrace_down_buffer_get(ESP_APPTRACE_DEST_TRAX, &sz,
↳100/*tmo in us*/);
if (ptr == NULL) {
    ESP_LOGE(TAG, "Failed to get buffer!");
    return ESP_FAIL;
}
if (sz > 4) {
    number = (uint32_t *)ptr;
    printf("Here is the number %d", *number);
} else {
    printf("No data");
}
esp_err_t res = esp_appttrace_down_buffer_put(ESP_APPTRACE_DEST_TRAX, ptr, 100/
↳*tmo in us*/);
if (res != ESP_OK) {
    /* in case of error host tracing tool (e.g. OpenOCD) will report
↳incomplete user buffer */
    ESP_LOGE(TAG, "Failed to put buffer!");
    return res;
}

```

2. The next step is to build the program image and download it to the target as described in the [Getting Started Guide](#).
3. Run OpenOCD (see [JTAG Debugging](#)).
4. Connect to OpenOCD telnet server. It can be done using the following command in terminal telnet <oocd_host> 4444. If telnet session is opened on the same machine which runs OpenOCD you can use localhost as <oocd_host> in the command above.
5. Start trace data collection using special OpenOCD command. This command will transfer tracing data and redirect them to specified file or socket (currently only files are supported as trace data destination). For description of the corresponding commands see [OpenOCD Application Level Tracing Commands](#).
6. The final step is to process received data. Since format of data is defined by user the processing stage is out of the scope of this document. Good starting points for data processor are python scripts in \$IDF_PATH/tools/esp_app_trace: appttrace_proc.py (used for feature tests) and logtrace_proc.py (see more details in section [Logging to Host](#)).

OpenOCD Application Level Tracing Commands HW UP BUFFER is shared between user data blocks and filling of the allocated memory is performed on behalf of the API caller (in task or ISR context). In multithreading environment it can happen that task/ISR which fills the buffer is preempted by another high priority task/ISR. So

it is possible situation that user data preparation process is not completed at the moment when that chunk is read by the host. To handle such conditions tracing module prepends all user data chunks with header which contains allocated user buffer size (2 bytes) and length of actually written data (2 bytes). So total length of the header is 4 bytes. OpenOCD command which reads trace data reports error when it reads incomplete user data chunk, but in any case it puts contents of the whole user chunk (including unfilled area) to output file.

Below is the description of available OpenOCD application tracing commands.

Note: Currently OpenOCD does not provide commands to send arbitrary user data to the target.

Command usage:

```
esp32 appttrace [start <options>] | [stop] | [status] | [dump <cores_num>
<outfile>]
```

Sub-commands:

start Start tracing (continuous streaming).
stop Stop tracing.
status Get tracing status.
dump Dump all data from (post-mortem dump).

Start command syntax:

```
start <outfile> [poll_period [trace_size [stop_tmo [wait4halt
[skip_size]]]]]
```

outfile Path to file to save data from both CPUs. This argument should have the following format: `file://path/to/file`.

poll_period Data polling period (in ms) for available trace data. If greater than 0 then command runs in non-blocking mode. By default 1 ms.

trace_size Maximum size of data to collect (in bytes). Tracing is stopped after specified amount of data is received. By default -1 (trace size stop trigger is disabled).

stop_tmo Idle timeout (in sec). Tracing is stopped if there is no data for specified period of time. By default -1 (disable this stop trigger). Optionally set it to value longer than longest pause between tracing commands from target.

wait4halt If 0 start tracing immediately, otherwise command waits for the target to be halted (after reset, by breakpoint etc.) and then automatically resumes it and starts tracing. By default 0.

skip_size Number of bytes to skip at the start. By default 0.

Note: If `poll_period` is 0, OpenOCD telnet command line will not be available until tracing is stopped. You must stop it manually by resetting the board or pressing Ctrl+C in OpenOCD window (not one with the telnet session). Another option is to set `trace_size` and wait until this size of data is collected. At this point tracing stops automatically.

Command usage examples:

1. Collect 2048 bytes of tracing data to a file “trace.log”. The file will be saved in “openocd-esp32” directory.

```
esp32 appttrace start file://trace.log 1 2048 5 0 0
```

The tracing data will be retrieved and saved in non-blocking mode. This process will stop automatically after 2048 bytes are collected, or if no data are available for more than 5 seconds.

Note: Tracing data is buffered before it is made available to OpenOCD. If you see “Data timeout!” message, then the target is likely sending not enough data to empty the buffer to OpenOCD before expiration of timeout. Either increase the timeout or use a function `esp_appttrace_flush()` to flush the data on specific intervals.

2. Retrieve tracing data indefinitely in non-blocking mode.


```
esp32 appttrace start file://trace.log 1 -1 -1 0 0
```

There is no limitation on the size of collected data and there is no any data timeout set. This process may be stopped by issuing `esp32 appttrace stop` command on OpenOCD telnet prompt, or by pressing Ctrl+C in OpenOCD window.

3. Retrieve tracing data and save them indefinitely.

```
esp32 appttrace start file://trace.log 0 -1 -1 0 0
```

OpenOCD telnet command line prompt will not be available until tracing is stopped. To stop tracing press Ctrl+C in OpenOCD window.

4. Wait for target to be halted. Then resume target's operation and start data retrieval. Stop after collecting 2048 bytes of data:

```
esp32 appttrace start file://trace.log 0 2048 -1 1 0
```

To configure tracing immediately after reset use the `openocd reset halt` command.

Logging to Host

IDF implements useful feature: logging to host via application level tracing library. This is a kind of semihosting when all `ESP_LOGx` calls send strings to be printed to the host instead of UART. This can be useful because “printing to host” eliminates some steps performed when logging to UART. The most part of work is done on the host.

By default IDF's logging library uses `vprintf`-like function to write formatted output to dedicated UART. In general it involves the following steps:

1. Format string is parsed to obtain type of each argument.
2. According to its type every argument is converted to string representation.
3. Format string combined with converted arguments is sent to UART.

Though implementation of `vprintf`-like function can be optimized to a certain level, all steps above have to be performed in any case and every step takes some time (especially item 3). So it frequently occurs that with additional log added to the program to identify the problem, the program behavior is changed and the problem cannot be reproduced or in the worst cases the program cannot work normally at all and ends up with an error or even hangs.

Possible ways to overcome this problem are to use higher UART bitrates (or another faster interface) and/or move string formatting procedure to the host.

Application level tracing feature can be used to transfer log information to host using `esp_appttrace_vprintf` function. This function does not perform full parsing of the format string and arguments, instead it just calculates number of arguments passed and sends them along with the format string address to the host. On the host log data are processed and printed out by a special Python script.

Limitations Current implementation of logging over JTAG has some limitations:

1. Tracing from `ESP_EARLY_LOGx` macros is not supported.
2. No support for `printf` arguments which size exceeds 4 bytes (e.g. `double` and `uint64_t`).
3. Only strings from `.rodata` section are supported as format strings and arguments.
4. Maximum number of `printf` arguments is 256.

How To Use It In order to use logging via trace module user needs to perform the following steps:

1. On target side special `vprintf`-like function needs to be installed. As it was mentioned earlier this function is `esp_appttrace_vprintf`. It sends log data to the host. Example code is provided in [system/app_trace_to_host](#).
2. Follow instructions in items 2-5 in [Application Specific Tracing](#).
3. To print out collected log records, run the following command in terminal: `$IDF_PATH/tools/esp_app_trace/logtrace_proc.py /path/to/trace/file /path/to/program/elf/file`.

Log Trace Processor Command Options Command usage:

```
logtrace_proc.py [-h] [--no-errors] <trace_file> <elf_file>
```

Positional arguments:

trace_file Path to log trace file**elf_file** Path to program ELF file

Optional arguments:

-h, --help show this help message and exit**--no-errors, -n** Do not print errors**System Behavior Analysis with SEGGER SystemView**

Another useful IDF feature built on top of application tracing library is the system level tracing which produces traces compatible with SEGGER SystemView tool (see [SystemView](#)). SEGGER SystemView is a real-time recording and visualization tool that allows to analyze runtime behavior of an application.

Note: Currently IDF-based application is able to generate SystemView compatible traces in form of files to be opened in SystemView application. The tracing process cannot yet be controlled using that tool.

How To Use It Support for this feature is enabled by *Component config > Application Level Tracing > FreeRTOS SystemView Tracing (CONFIG_SYSVIEW_ENABLE)* menuconfig option. There are several other options enabled under the same menu:

1. *ESP32 timer to use as SystemView timestamp source (CONFIG_SYSVIEW_TS_SOURCE)* selects the source of timestamps for SystemView events. In single core mode timestamps are generated using ESP32 internal cycle counter running at maximum 240 Mhz (~4 ns granularity). In dual-core mode external timer working at 40 Mhz is used, so timestamp granularity is 25 ns.
2. Individually enabled or disabled collection of SystemView events (CONFIG_SYSVIEW_EVT_XXX):
 - Trace Buffer Overflow Event
 - ISR Enter Event
 - ISR Exit Event
 - ISR Exit to Scheduler Event
 - Task Start Execution Event
 - Task Stop Execution Event
 - Task Start Ready State Event
 - Task Stop Ready State Event
 - Task Create Event
 - Task Terminate Event
 - System Idle Event
 - Timer Enter Event
 - Timer Exit Event

IDF has all the code required to produce SystemView compatible traces, so user can just configure necessary project options (see above), build, download the image to target and use OpenOCD to collect data as described in the previous sections.

OpenOCD SystemView Tracing Command Options Command usage:

```
esp32 sysview [start <options>] | [stop] | [status]
```

Sub-commands:

start Start tracing (continuous streaming).**stop** Stop tracing.**status** Get tracing status.

Start command syntax:

```
start <outfile1> [outfile2] [poll_period [trace_size [stop_tmo]]]
```

outfile1 Path to file to save data from PRO CPU. This argument should have the following format: `file://path/to/file`.

outfile2 Path to file to save data from APP CPU. This argument should have the following format: `file://path/to/file`.

poll_period Data polling period (in ms) for available trace data. If greater than 0 then command runs in non-blocking mode. By default 1 ms.

trace_size Maximum size of data to collect (in bytes). Tracing is stopped after specified amount of data is received. By default -1 (trace size stop trigger is disabled).

stop_tmo Idle timeout (in sec). Tracing is stopped if there is no data for specified period of time. By default -1 (disable this stop trigger).

Note: If `poll_period` is 0 OpenOCD telnet command line will not be available until tracing is stopped. You must stop it manually by resetting the board or pressing Ctrl+C in OpenOCD window (not one with the telnet session). Another option is to set `trace_size` and wait until this size of data is collected. At this point tracing stops automatically.

Command usage examples:

1. Collect SystemView tracing data to files “pro-cpu.SVdat” and “app-cpu.SVdat” . The files will be saved in “openocd-esp32” directory.

```
esp32 sysview start file://pro-cpu.SVdat file://app-cpu.SVdat
```

The tracing data will be retrieved and saved in non-blocking mode. To stop data this process enter `esp32 apptrace stop` command on OpenOCD telnet prompt, optionally pressing Ctrl+C in OpenOCD window.

2. Retrieve tracing data and save them indefinitely.

```
esp32 sysview start file://pro-cpu.SVdat file://app-cpu.SVdat 0 -1 -1
```

OpenOCD telnet command line prompt will not be available until tracing is stopped. To stop tracing, press Ctrl+C in OpenOCD window.

Data Visualization After trace data are collected user can use special tool to visualize the results and inspect behavior of the program. Unfortunately SystemView does not support tracing from multiple cores. So when tracing from ESP32 working in dual-core mode two files are generated: one for PRO CPU and another one for APP CPU. User can load every file into separate instance of the tool.

It is uneasy and awkward to analyze data for every core in separate instance of the tool. Fortunately there is Eclipse plugin called *Impulse* which can load several trace files and makes it possible to inspect events from both cores in one view. Also this plugin has no limitation of 1,000,000 events as compared to free version of SystemView.

Good instruction on how to install, configure and visualize data in Impulse from one core can be found [here](#).

Note: IDF uses its own mapping for SystemView FreeRTOS events IDs, so user needs to replace original file with mapping `$(SYSVIEW_INSTALL_DIR)/Description/SYSVIEW_FreeRTOS.txt` with `$(IDF_PATH)/docs/api-guides/SYSVIEW_FreeRTOS.txt`. Also contents of that IDF specific file should be used when configuring SystemView serializer using above link.

Configure Impulse for Dual Core Traces After installing Impulse and ensuring that it can successfully load trace files for each core in separate tabs user can add special Multi Adapter port and load both files into one view. To do this user needs to do the following in Eclipse:

1. Open ‘Signal Ports’ view. Go to Windows->Show View->Other menu. Find ‘Signal Ports’ view in Impulse folder and double-click on it.
2. In ‘Signal Ports’ view right-click on ‘Ports’ and select ‘Add ...’ ->New Multi Adapter Port

3. In open dialog Press 'Add' button and select 'New Pipe/File' .
4. In open dialog select 'SystemView Serializer' as Serializer and set path to PRO CPU trace file. Press OK.
5. Repeat steps 3-4 for APP CPU trace file.
6. Double-click on created port. View for this port should open.
7. Click Start/Stop Streaming button. Data should be loaded.
8. Use 'Zoom Out' , 'Zoom In' and 'Zoom Fit' button to inspect data.
9. For settings measurement cursors and other features please see [Impulse documentation](#)).

Note: If you have problems with visualization (no data are shown or strange behavior of zoom action is observed) you can try to delete current signal hierarchy and double click on the necessary file or port. Eclipse will ask you to create new signal hierarchy.

Gcov (Source Code Coverage)

Basics of Gcov and Lcov Source code coverage is data indicating the count and frequency of every program execution path that has been taken within a program' s runtime. **Gcov** is a GCC tool that, when used in concert with the compiler, can generate log files indicating the execution count of each line of a source file. The **Lcov** tool is similar to Gcov but is a graphical front end for Gcov, and generates code coverage reports in HTML format.

Generally, using Gcov to compile and run programs on the Host will undergo these steps:

1. Compile the source code using GCC with the `--coverage` option enabled. This will cause the compiler to generate a `.gcno` notes files during compilation. The notes files contain information to reconstruct execution path block graphs and map each block to source code line numbers. Each source file compiled with the `--coverage` option should have their own `.gcno` file of the same name (e.g., a `main.c` will generate a `main.gcno` when compiled).
2. Execute the program. During execution, the program should generate `.gcda` data files. These data files contain the counts of the number of times an execution path was taken. The program will generate a `.gcda` file for each source file compiled with the `--coverage` option (e.g., `main.c` will generate a `main.gcda`).
3. Gcov or Lcov can be used generate a code coverage based on the `.gcno`, `.gcda`, and source files. Gcov will generate a text based coverage report for each source file in the form of a `.gcov` file, whilst Lcov will generate a coverage report in HTML format.

Gcov and Lcov in ESP-IDF Using Gcov in ESP-IDF is complicated by the fact that the program is running remotely from the Host (i.e., on the target). The code coverage data (i.e., the `.gcda` files) is initially stored on the target itself. OpenOCD is then used to dump the code coverage data from the target to the host via JTAG during runtime. Using Gcov in ESP-IDF can be split into the following steps.

1. [Setting Up a Project for Gcov](#)
2. [Dumping Code Coverage Data](#)
3. [Generating Coverage Report](#)

Setting Up a Project for Gcov

Compiler Option In order to obtain code coverage data in a project, one or more source files within the project must be compiled with the `--coverage` option. In ESP-IDF, this can be achieved at the component level or the individual source file level:

To cause all source files in a component to be compiled with the `--coverage` option.

- Add `target_compile_options(${COMPONENT_LIB} PRIVATE --coverage)` to the `CMakeLists.txt` file of the component if using CMake.
- Add `CFLAGS += --coverage` to the `component.mk` file of the component if using Make.

To cause a select number of source files (e.g. `source1.c` and `source2.c`) in the same component to be compiled with the

- Add `set_source_files_properties(source1.c source2.c PROPERTIES COMPILE_FLAGS --coverage)` to the `CMakeLists.txt` file of the component if using CMake.

- Add `source1.o: CFLAGS += --coverage` and `source2.o: CFLAGS += --coverage` to the component `.mk` file of the component if using Make.

When a source file is compiled with the `--coverage` option (e.g. `gcov_example.c`), the compiler will generate the `gcov_example.gcno` file in the project's build directory.

Project Configuration Before building a project with source code coverage, ensure that the following project configuration options are enabled by running `idf.py menuconfig` (or `make menuconfig` if using the legacy Make build system).

- Enable the application tracing module by choosing *Trace Memory* for the `CONFIG_APPTRACE_DESTINATION` option.
- Enable Gcov to host via the `CONFIG_APPTRACE_GCOV_ENABLE`

Dumping Code Coverage Data Once a project has been compiled with the `--coverage` option and flashed onto the target, code coverage data will be stored internally on the target (i.e., in trace memory) whilst the application runs. The process of transferring code coverage data from the target to the Host is known as dumping.

The dumping of coverage data is done via OpenOCD (see *JTAG Debugging* on how to setup and run OpenOCD). A dump is triggered by issuing commands to OpenOCD, therefore a telnet session to OpenOCD must be opened to issue such commands (run `telnet localhost 4444`). Note that GDB could be used instead of telnet to issue commands to OpenOCD, however all commands issued from GDB will need to be prefixed as `mon <occd_command>`.

When the target dumps code coverage data, the `.gcda` files are stored in the project's build directory. For example, if `gcov_example_main.c` of the main component was compiled with the `--coverage` option, then dumping the code coverage data would generate a `gcov_example_main.gcda` in `build/esp-idf/main/CMakeFiles/__idf_main.dir/gcov_example_main.c.gcda` (or `build/main/gcov_example_main.gcda` if using the legacy Make build system). Note that the `.gcno` files produced during compilation are also placed in the same directory.

The dumping of code coverage data can be done multiple times throughout an application's life time. Each dump will simply update the `.gcda` file with the newest code coverage information. Code coverage data is accumulative, thus the newest data will contain the total execution count of each code path over the application's entire lifetime.

ESP-IDF supports two methods of dumping code coverage data from the target to the host:

- Instant Run-Time Dump
- Hard-coded Dump

Instant Run-Time Dump An Instant Run-Time Dump is triggered by calling the `esp32 gcov OpenOCD` command (via a telnet session). Once called, OpenOCD will immediately preempt the ESP32's current state and execute a builtin IDF Gcov debug stub function. The debug stub function will handle the dumping of data to the Host. Upon completion, the ESP32 will resume its current state.

Note: Due to the use of the debug stub function, the OpenOCD Debug Stub option must be enabled in project configuration. The option can be found under `Component config -> ESP32-specific -> OpenOCD debug stubs`.

Hard-coded Dump A Hard-coded Dump is triggered by the application itself by calling `esp_gcov_dump()` from somewhere within the application. When called, the application will halt and wait for OpenOCD to connect and retrieve the code coverage data. Once `esp_gcov_dump()` is called, the Host must execute the `esp32 gcov dump OpenOCD` command (via a telnet session). The `esp32 gcov dump` command will cause OpenOCD to connect to the ESP32, retrieve the code coverage data, then disconnect from the ESP32 thus allowing the application to resume. Hard-coded Dumps can also be triggered multiple times throughout an application's lifetime.

Hard-coded dumps are useful if code coverage data is required at certain points of an application's lifetime by placing `esp_gcov_dump()` where necessary (e.g., after application initialization, during each iteration of an application's main loop).

GDB can be used to set a breakpoint on `esp_gcov_dump()`, then call `mon esp32 gcov dump` automatically via the use a `gdbinit` script (see Using GDB from [Command Line](#)).

The following GDB script will add a breakpoint at `esp_gcov_dump()`, then call the `mon esp32 gcov dump` OpenOCD command.

```
b esp_gcov_dump
commands
mon esp32 gcov dump
end
```

Note: Note that all OpenOCD commands should be invoked in GDB as: `mon <occd_command>`.

Generating Coverage Report Once the code coverage data has been dumped, the `.gcno`, `.gcda` and the source files can be used to generate a code coverage report. A code coverage report is simply a report indicating the number of times each line in a source file has been executed.

Both Gcov and Lcov (along with `genhtml`) can be used to generate code coverage reports. Gcov is provided along with the Xtensa toolchain, whilst Lcov may need to be installed separately. For details on how to use Gcov or Lcov, refer to [Gcov documentation](#) and [Lcov documentation](#).

Note: There is currently no support for Lcov for the CMake build system on Windows.

Adding Lcov Build Target to Project To make report generation more convenient, users can define additional build targets in their projects such report generation can be done with a single build command.

CMake Build System For the CMake build systems, add the following lines to the `CMakeLists.txt` file of your project.

```
include($ENV{IDF_PATH}/tools/cmake/gcov.cmake)
idf_create_coverage_report(${CMAKE_CURRENT_BINARY_DIR}/coverage_report)
idf_clean_coverage_report(${CMAKE_CURRENT_BINARY_DIR}/coverage_report)
```

The following commands can now be used:

- `cmake --build build/ --target lcov-report` will generate an HTML coverage report in `$(BUILD_DIR_BASE)/coverage_report/html` directory.
- `cmake --build build/ --target cov-data-clean` will remove all coverage data files and report.

Make Build System For the Make build systems, add the following lines to the Makefile of your project.

```
GCOV := $(call dequote,$(CONFIG_SDK_TOOLPREFIX))gcov
REPORT_DIR := $(BUILD_DIR_BASE)/coverage_report

lcov-report:
    echo "Generating coverage report in: $(REPORT_DIR) "
    echo "Using gcov: $(GCOV) "
    mkdir -p $(REPORT_DIR)/html
    lcov --gcov-tool $(GCOV) -c -d $(BUILD_DIR_BASE) -o $(REPORT_DIR)/$(PROJECT_
↵NAME).info
    genhtml -o $(REPORT_DIR)/html $(REPORT_DIR)/$(PROJECT_NAME).info

cov-data-clean:
    echo "Remove coverage data files..."
```

(continues on next page)

(continued from previous page)

```
find $(BUILD_DIR_BASE) -name "*.gcda" -exec rm {} +
rm -rf $(REPORT_DIR)

.PHONY: lcov-report cov-data-clean
```

The following commands can now be used:

- `make lcov-report` will generate an HTML coverage report in `$(BUILD_DIR_BASE)/coverage_report/html` directory.
- `make cov-data-clean` will remove all coverage data files and report.

5.2 BluFi

5.2.1 Overview

The BluFi for ESP32 is a Wi-Fi network configuration function via Bluetooth channel. It provides a secure protocol to pass Wi-Fi configuration and credentials to the ESP32. Using this information ESP32 can then e.g. connect to an AP or establish a SoftAP.

Fragmenting, data encryption, checksum verification in the BluFi layer are the key elements of this process.

You can customize symmetric encryption, asymmetric encryption and checksum support customization. Here we use the DH algorithm for key negotiation, 128-AES algorithm for data encryption, and CRC16 algorithm for checksum verification.

5.2.2 The BluFi Flow

The BluFi networking flow includes the configuration of the SoftAP and Station.

The following uses Station as an example to illustrate the core parts of the procedure, including broadcast, connection, service discovery, negotiation of the shared key, data transmission, connection status backhaul.

1. Set the ESP32 into GATT Server mode and then it will send broadcasts with specific *advertising data*. You can customize this broadcast as needed, which is not a part of the BluFi Profile.
2. Use the App installed on the mobile phone to search for this particular broadcast. The mobile phone will connect to ESP32 as the GATT Client once the broadcast is confirmed. The App used during this part is up to you.
3. After the GATT connection is successfully established, the mobile phone will send a data frame for key negotiation to ESP32 (see the section *The Frame Formats Defined in BluFi* for details).
4. After ESP32 receives the data frame of key negotiation, it will parse the content according to the user-defined negotiation method.
5. The mobile phone works with ESP32 for key negotiation using the encryption algorithms such as DH, RSA or ECC.
6. After the negotiation process is completed, the mobile phone will send a control frame for security-mode setup to ESP32.
7. When receiving this control frame, ESP32 will be able to encrypt and decrypt the communication data using the shared key and the security configuration.
8. The mobile phone sends the data frame defined in the section of *The Frame Formats Defined in BluFi*, with the Wi-Fi configuration information to ESP32, including SSID, password, etc.
9. The mobile phone sends a control frame of Wi-Fi connection request to ESP32. When receiving this control frame, ESP32 will regard the communication of essential information as done and get ready to connect to the Wi-Fi.
10. After connecting to the Wi-Fi, ESP32 will send a control frame of Wi-Fi connection status report to the mobile phone, to report the connection status. At this point the networking procedure is completed.

Note:

1. After ESP32 receives the control frame of security-mode configuration, it will execute the operations in accordance with the defined security mode.
2. The data lengths before and after symmetric encryption/decryption must stay the same. It also supports in-place encryption and decryption.

5.2.3 The flow chart of BluFi

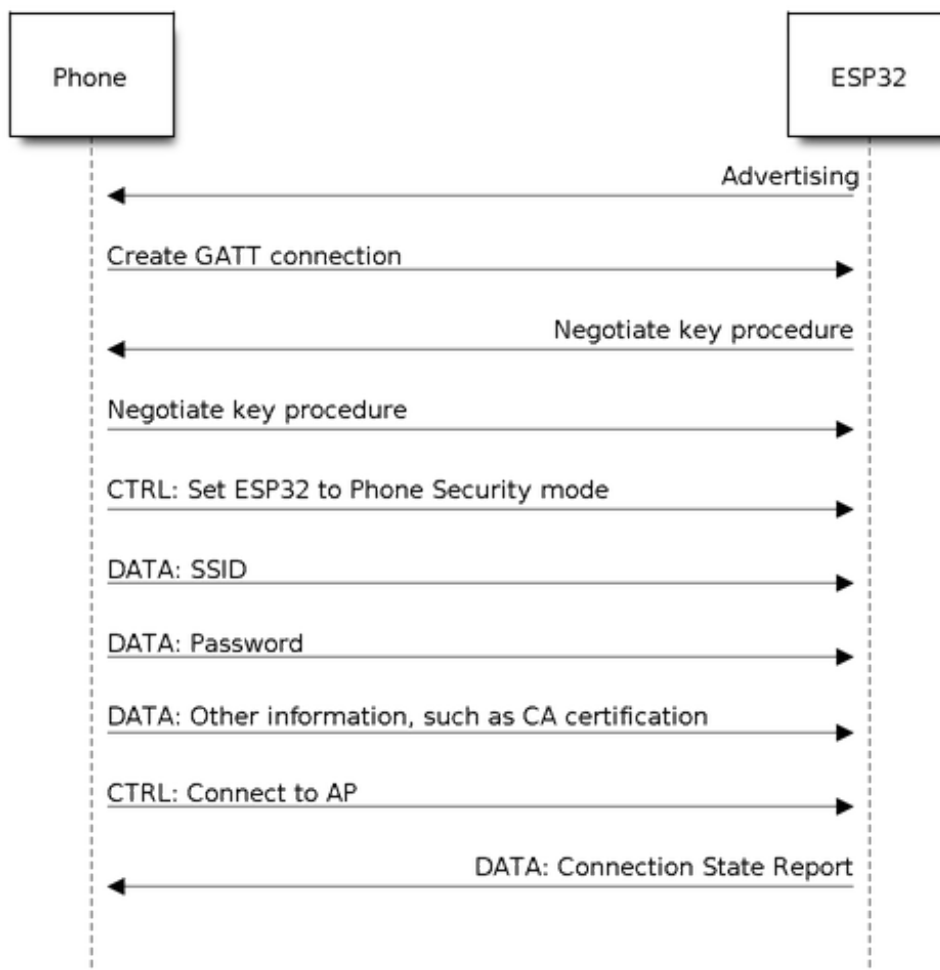


Fig. 2: BluFi Flow Chart

5.2.4 The Frame Formats Defined in BluFi

The frame formats for the communication between the mobile phone App and ESP32 are defined as follows:

The frame format with no fragment (8 bit):

| Description | Value |
|-----------------|-----------------|
| LSB - Type | 1 |
| Frame Control | 1 |
| Sequence Number | 1 |
| Data Length | 1 |
| Data | \${Data Length} |
| MSB - CheckSum | 2 |

If the **Frame Ctrl** bit is enabled, the **Total length** bit indicates the length of remaining part of the frame. It can tell the remote how much memory needs to be allocated.

The frame format with fragments (8 bit) :

| Description | Value | |
|--------------------|----------------------|---------------------|
| LSB - Type | 1 | |
| FrameControl(Frag) | 1 | |
| SequenceNumber | 1 | |
| DataLength | 1 | |
| Data | Total Content Length | 2 |
| | Content | \${Data Length} - 2 |
| MSB - CheckSum | 2 | |

Normally, the control frame does not contain data bits, except for Ack Frame.

The format of Ack Frame (8 bit) :

| Description | Value | |
|------------------|-----------------------|---|
| LSB - Type (Ack) | 1 | |
| Frame Control | 1 | |
| SequenceNumber | 1 | |
| DataLength | 1 | |
| Data | Acked Sequence Number | 2 |
| | | |
| MSB - CheckSum | 2 | |

1. Type

The **Type** field, taking 1 byte, is divided into **Type** and **Subtype**, that Type uses the lower 2 bits and **Subtype** uses the upper 6 bits.

- The control frame is not encrypted for the time being and supports to be verified;
- The data frame supports to be encrypted and verified.

1.1 Control Frame (0x0 b' 00)

| Control Frame (Binary) | Implementation | Explanation | Note |
|------------------------|--|---|--|
| 0x0 (b' 000000) | Ack | The data field of the Ack frame uses the same sequence value of the frame to reply to. | The data field consumes a byte and its value is the same as the sequence field of the frame to reply to. |
| 0x1 (b' 000001) | Set ESP32 to the security mode. | To inform ESP32 of the security mode to use when sending data, which is allowed to be reset multiple times during the process. Each setting affects the subsequent security mode used. If it is not set, ESP32 will send the control frame and data frame with no checksum and encryption by default. The data transmission from the mobile phone to ESP32 is controlled by this control frame. | <p>The data field consumes a byte. The higher 4 bits are for the security mode setting of the control frame, and the lower 4 bits are for the security mode setting of the data frame.</p> <p>b' 0000: no checksum and no encryption;</p> <p>b' 0001: with checksum but no encryption;</p> <p>b' 0010: no checksum but with encryption;</p> <p>b' 0011: with both checksum and encryption.</p> |
| 0x2 (b' 000010) | Set the opmode of Wi-Fi. | The frame contains opmode settings for configuring for the Wi-Fi mode of ESP32. | <p>data[0] is for opmode settings, including:</p> <p>0x00: NULL;</p> <p>0x01: STA;</p> <p>0x02: SoftAP;</p> <p>0x03: SoftAP&STA.</p> <p>Please set the SSID/Password/Max Connection Number of the AP mode in the first place if an AP gets involved.</p> |
| 0x3 (b' 000011) | Connect ESP32 to the AP. | To notify ESP32 that the essential information has been sent and it is allowed to connect to the AP. | No data field is contained. |
| 0x4 (b' 000100) | Disconnect ESP32 from the AP. | | No data field is contained. |
| 0x5 (b' 000101) | To get the information of ESP32's Wi-Fi mode and its status. | | No data field is contained. When receiving this control frame, ESP32 will send back a follow-up frame of Wi-Fi connection state report to the mobile phone with the information of the current opmode, connection status, SSID and so on. The types of information sent to the mobile phone is defined by the application installed on the phone. |

| | | | |
|-----------------|------------|--|--|
| 0x6 (b' 000110) | Disconnect | | Date[0~5] is taken as the MAC address for the STA device. If there is no STA device... |
|-----------------|------------|--|--|

1.2 Data Frame (0x1 b' 01)

| Data Frame(Binary) | Implication | Explanation |
|--------------------|--|---|
| 0x0 (b' 000000) | Send the negotiation data. | The negotiation data will be sent to the callback |
| 0x1 (b' 000001) | Send the BSSID for STA mode. | To send the BSSID of the AP for the STA device |
| 0x2 (b' 000010) | Send the SSID for STA mode. | To send the SSID of the AP for the STA device |
| 0x3 (b' 000011) | Send the password for STA mode. | To send the password of the AP for the STA device |
| 0x4 (b' 000100) | Send the SSID for SoftAP mode. | |
| 0x5 (b' 000101) | Send the password for SoftAP mode. | |
| 0x6 (b' 000110) | Set the maximum connection number for SoftAP mode. | |
| 0x7 (b' 000111) | Set the authentication mode for the SoftAP. | |
| 0x8 (b' 001000) | Set the channel amount for SoftAP mode. | |
| 0x9 (b' 001001) | Username | It provides the username of the GATT client when using encryption |
| 0xa (b' 001010) | CA Certification | It provides the CA Certification when using encryption |
| 0xb (b' 001011) | Client Certification | It provides the client certification when using encryption |
| 0xc (b' 001100) | Server Certification | It provides the server certification when using encryption |
| 0xd (b' 001101) | ClientPrivate Key | It provides the private key of the client when using encryption |
| 0xe (b' 001110) | ServerPrivate Key | It provides the private key of the server when using encryption |
| 0xf (b' 001111) | Wi-Fi Connection State Report | To notify the phone of the ESP32's Wi-Fi status, including STA status and SoftAP status. It is for the STA device to connect to the mobile phone or the SoftAP. However, when the mobile phone receives the Wi-Fi status, it can reply to other frames in addition to this frame. |
| 0x10 (b' 010000) | Version | |
| 0x11 (b' 010001) | Wi-Fi List | To send the Wi-Fi list to ESP32. |
| 0x12 (b' 010010) | Report Error | To notify the mobile phone that there is an error with BluFi. |
| 0x13 (b' 010011) | Custom Data | To send or receive custom data. |

2. Frame Control

Control field, takes 1 byte and each bit has a different meaning.

| Bit | Meaning |
|----------------------------|---|
| 0x01 | Indicates whether the frame is encrypted. |
| | 1 means encryption, and 0 means unencrypted. |
| | The encrypted part of the frame includes the full clear data before the DATA field is encrypted (no checksum). |
| | Control frame is not encrypted, so this bit is 0. |
| 0x02 | The data field that indicates whether a frame contains a checksum (such as SHA1, MD5, CRC, etc.) for the end of the frame data field includes SEQUCNE + data length + clear text. Both the control frame and the data frame can contain a check bit or not. |
| 0x04 | Represents the data direction. |
| | 0 means the mobile phone to ESP32; 1 means ESP32 to the mobile phone. |
| 0x08 | Indicates whether the other person is required to reply to an ACK. |
| | 0 indicates no requirement; 1 indicates to reply Ack. |
| 0x10 | Indicates whether there are subsequent data fragments. |
| | 0 indicates that there are no subsequent data fragments for this frame; 1 indicates that there are subsequent data fragments and used to transmit longer data. |
| | In the case of a frag frame, the total length of the current content section + subsequent content section is given, in the first 2 bytes of the data field (that is, the content data of the maximum support 64K). |
| 0x10~0x80 re- served | |

3. Sequence Control

Sequence control field. When a frame is sent, the value of sequence field is automatically incremented by 1 regardless of the type of frame, which prevents Replay Attack. The sequence is cleared after each reconnection.

4. Length

The length of the data field that does not include CheckSum.

5. Data

The instruction of the data field is different according to various values of Type or Subtype. Please refer to the table above.

6. CheckSum

This field takes 2 bytes that is used to check “sequence + data length + clear text data” .

5.2.5 The Security Implementation of ESP32

1. Securing data

To ensure that the transmission of the Wi-Fi SSID and password is secure, the message needs to be encrypted using symmetric encryption algorithms, such as AES, DES and so on. Before using symmetric encryption algorithms, the devices are required to negotiate (or generate) a shared key using an asymmetric encryption algorithm (DH, RSA, ECC, etc).

2. Ensuring data integrity

To ensure data integrity, you need to add a checksum algorithm, such as SHA1, MD5, CRC, etc.

3. Securing identity (signature)

Algorithm like RSA can be used to secure identity. But for DH, it needs other algorithms as a companion for signature.

4. Replay attack prevention

It is added to the Sequence field and used during the checksum verification.

For the coding of ESP32, you can determine and develop the security processing, such as key negotiation. The mobile application sends the negotiation data to ESP32 and then the data will be sent to the application layer for processing. If the application layer does not process it, you can use the DH encryption algorithm provided by BluFi to negotiate the key.

The application layer needs to register several security-related functions to BluFi:

```
typedef void (*esp_blufi_negotiate_data_handler_t)(uint8_t *data, int len, uint8_t *  
↪**output_data, int *output_len, bool *need_free)
```

This function is for ESP32 to receive normal data during negotiation, and after processing is completed, the data will be transmitted using Output_data and Output_len.

BluFi will send output_data from Negotiate_data_handler after Negotiate_data_handler is called.

Here are two “*” , because the length of the data to be emitted is unknown that requires the function to allocate itself (malloc) or point to the global variable, and to inform whether the memory needs to be freed by NEED_FREE.

```
typedef int (* esp_blufi_encrypt_func_t)(uint8_t iv8, uint8_t *crypt_data, int_  
↪crypt_len)
```

The data to be encrypted and decrypted must use the same length. The IV8 is a 8 bit sequence value of frames, which can be used as a 8 bit of IV.

```
typedef int (* esp_blufi_decrypt_func_t)(uint8_t iv8, uint8_t *crypt_data, int_  
↪crypt_len)
```

The data to be encrypted and decrypted must use the same length. The IV8 is a 8 bit sequence value of frames, which can be used as a 8 bit of IV.

```
typedef uint16_t (*esp_blufi_checksum_func_t)(uint8_t iv8, uint8_t *data, int len)
```

This function is used to compute CheckSum and return a value of CheckSum. BluFi uses the returned value to compare the CheckSum of the frame.

5.2.6 GATT Related Instructions

UUID

BluFi Service UUID: 0xFFFF, 16 bit

BluFi (the mobile -> ESP32): 0xFF01, writable

Blufi (ESP32 -> the mobile phone): 0xFF02, readable and callable

5.3 Bootloader

Bootloader performs the following functions:

1. Minimal initial configuration of internal modules;
2. Select the application partition to boot, based on the partition table and ota_data (if any);
3. Load this image to RAM (IRAM & DRAM) and transfer management to it.

Bootloader is located at the address *0x1000* in the flash.

5.3.1 Bootloader compatibility

It is recommended to update to newer *versions of ESP-IDF*: when they are released. The OTA (over the air) update process can flash new apps in the field but cannot flash a new bootloader. For this reason, the bootloader supports booting apps built from newer versions of ESP-IDF.

The bootloader does not support booting apps from older versions of ESP-IDF. When updating ESP-IDF manually on an existing product that might need to downgrade the app to an older version, keep using the older ESP-IDF bootloader binary as well.

Note: If testing an OTA update for an existing product in production, always test it using the same ESP-IDF bootloader binary that is deployed in production.

Before ESP-IDF V2.1

Bootloaders built from very old versions of ESP-IDF (before ESP-IDF V2.1) perform less hardware configuration than newer versions. When using a bootloader from these early ESP-IDF versions and building a new app, enable the config option `CONFIG_ESP32_COMPATIBLE_PRE_V2_1_BOOTLOADERS`.

Before ESP-IDF V3.1

Bootloaders built from versions of ESP-IDF before V3.1 do not support MD5 checksums in the partition table binary. When using a bootloader from these ESP-IDF versions and building a new app, enable the config option `CONFIG_ESP32_COMPATIBLE_PRE_V3_1_BOOTLOADERS`.

SPI Flash Configuration

Each ESP-IDF application or bootloader .bin file contains a header with `CONFIG_ESPTOOLPY_FLASHMODE`, `CONFIG_ESPTOOLPY_FLASHFREQ`, `CONFIG_ESPTOOLPY_FLASHSIZE` embedded in it. These are used to configure the SPI flash during boot.

The :first stage bootloader in ROM reads the second stage bootloader header from flash and uses these settings to load it. However, at this time the system clock speed is lower than configured and not all flash modes are supported. When the second stage bootloader then runs and re-configures the flash, it reads values from the currently selected app binary header not the bootloader header. This allows an OTA update to change the SPI flash settings in use.

Bootloaders prior to ESP-IDF V4.0 used the bootloader's own header to configure the SPI flash, meaning these values could not be changed in an update. To maintain compatibility with older bootloaders, the app re-initializes the flash settings during app startup using the configuration found in the app header.

5.3.2 FACTORY reset

The user can write a basic working firmware and load it into the factory partition. Next, update the firmware via OTA (over the air). The updated firmware will be loaded into an OTA app partition slot and the OTA data partition is updated to boot from this partition. If you want to be able to roll back to the factory firmware and clear the settings, then you need to set `CONFIG_BOOTLOADER_FACTORY_RESET`. The factory reset mechanism allows to reset the device to factory settings:

- Clear one or more data partitions.
- Boot from “factory” partition.

`CONFIG_BOOTLOADER_DATA_FACTORY_RESET` allows customers to select which data partitions will be erased when the factory reset is executed. Can specify the names of partitions through comma-delimited with optional spaces for readability. (Like this: “nvs, phy_init, nvs_custom, ..”). Make sure that the name specified in the partition table and here are the same. Partitions of type “app” cannot be specified here.

`CONFIG_BOOTLOADER_OTA_DATA_ERASE` - the device will boot from “factory” partition after a factory reset. The OTA data partition will be cleared.

`CONFIG_BOOTLOADER_NUM_PIN_FACTORY_RESET` - number of the GPIO input for factory reset uses to trigger a factory reset, this GPIO must be pulled low on reset to trigger this.

`CONFIG_BOOTLOADER_HOLD_TIME_GPIO` - this is hold time of GPIO for reset/test mode (by default 5 seconds). The GPIO must be held low continuously for this period of time after reset before a factory reset or test partition boot (as applicable) is performed.

Partition table.:

```
# Name, Type, SubType, Offset, Size, Flags
# Note: if you have increased the bootloader size, make sure to update the offsets.
↳to avoid overlap
nvs, data, nvs, 0x9000, 0x4000
otadata, data, ota, 0xd000, 0x2000
phy_init, data, phy, 0xf000, 0x1000
factory, 0, 0, 0x10000, 1M
test, 0, test, , 512K
ota_0, 0, ota_0, , 512K
ota_1, 0, ota_1, , 512K
```

5.3.3 Boot from TEST firmware

The user can write a special firmware for testing in production, and run it as needed. The partition table also needs a dedicated partition for this testing firmware (See *partition table*). To trigger a test app you need to set `CONFIG_BOOTLOADER_APP_TEST`.

`CONFIG_BOOTLOADER_NUM_PIN_APP_TEST` - GPIO number to boot TEST partition. The selected GPIO will be configured as an input with internal pull-up enabled. To trigger a test app, this GPIO must be pulled low on reset. After the GPIO input is deactivated and the device reboots, the normally configured application will boot (factory or any OTA slot).

`CONFIG_BOOTLOADER_HOLD_TIME_GPIO` - this is hold time of GPIO for reset/test mode (by default 5 seconds). The GPIO must be held low continuously for this period of time after reset before a factory reset or test partition boot (as applicable) is performed.

5.3.4 Fast boot from Deep Sleep

The bootloader has the `CONFIG_BOOTLOADER_SKIP_VALIDATE_IN_DEEP_SLEEP` option which allows to reduce the wake-up time (useful to reduce consumption). This option is available when the `CONFIG_SECURE_BOOT` option is disabled. Reduction of time is achieved due to the lack of image verification. During the first boot, the bootloader stores the address of the application being launched in the RTC FAST memory. And during the awakening, this address is used for booting without any checks, thus fast loading is achieved.

5.3.5 Customer bootloader

The current bootloader implementation allows the customer to override it. To do this, you must copy the folder `/esp-idf/components/bootloader` and then edit `/your_project/components/bootloader/subproject/main/bootloader_main.c`. In the bootloader space, you can not use the drivers and functions from other components. If necessary, then the required functionality should be placed in the folder `bootloader` (note that this will increase its size). It is necessary to monitor its size because there can be overlays in memory with a partition table leading to damage. At the moment the bootloader is limited to the partition table from the address `0x8000`.

5.4 Build System

This document explains the implementation of the ESP-IDF build system and the concept of “components”. Read this document if you want to know how to organize and build a new ESP-IDF project or component.

Note: This document describes the CMake-based build system, which is the default since ESP-IDF V4.0. ESP-IDF also supports a *legacy build system based on GNU Make*, which was the default before ESP-IDF V4.0.

5.4.1 Overview

An ESP-IDF project can be seen as an amalgamation of a number of components. For example, for a webserver that shows the current humidity, there could be:

- The ESP32 base libraries (libc, ROM bindings, etc)
- The Wi-Fi drivers
- A TCP/IP stack
- The FreeRTOS operating system
- A webserver
- A driver for the humidity sensor
- Main code tying it all together

ESP-IDF makes these components explicit and configurable. To do that, when a project is compiled, the build system will look up all the components in the ESP-IDF directories, the project directories and (optionally) in additional custom component directories. It then allows the user to configure the ESP-IDF project using a text-based menu system to customize each component. After the components in the project are configured, the build system will compile the project.

Concepts

- A “project” is a directory that contains all the files and configuration to build a single “app” (executable), as well as additional supporting elements such as a partition table, data/filesystem partitions, and a bootloader.
- “Project configuration” is held in a single file called `sdkconfig` in the root directory of the project. This configuration file is modified via `idf.py menuconfig` to customise the configuration of the project. A single project contains exactly one project configuration.
- An “app” is an executable which is built by ESP-IDF. A single project will usually build two apps - a “project app” (the main executable, ie your custom firmware) and a “bootloader app” (the initial bootloader program which launches the project app).
- “components” are modular pieces of standalone code which are compiled into static libraries (.a files) and linked into an app. Some are provided by ESP-IDF itself, others may be sourced from other places.
- “Target” is the hardware for which an application is built. At the moment, ESP-IDF supports `esp32` and `esp32s2beta` targets.

Some things are not part of the project:

- “ESP-IDF” is not part of the project. Instead it is standalone, and linked to the project via the `IDF_PATH` environment variable which holds the path of the `esp-idf` directory. This allows the IDF framework to be decoupled from your project.
- The toolchain for compilation is not part of the project. The toolchain should be installed in the system command line `PATH`.

5.4.2 Using the Build System

`idf.py`

The `idf.py` command line tool provides a front-end for easily managing your project builds. It manages the following tools:

- [CMake](#), which configures the project to be built
- A command line build tool (either [Ninja](#) build or *GNU Make*)
- [esptool.py](#) for flashing ESP32.

The [getting started guide](#) contains a brief introduction to how to set up `idf.py` to configure, build, and flash projects.

`idf.py` should be run in an ESP-IDF “project” directory, ie one containing a `CMakeLists.txt` file. Older style projects with a `Makefile` will not work with `idf.py`.

Type `idf.py --help` for a list of commands. Here are a summary of the most useful ones:

- `idf.py set-target <target>` sets the target (chip) for which the project is built. See [Selecting the Target](#).
- `idf.py menuconfig` runs the “menuconfig” tool to configure the project.
- `idf.py build` will build the project found in the current directory. This can involve multiple steps:
 - Create the build directory if needed. The sub-directory `build` is used to hold build output, although this can be changed with the `-B` option.
 - Run [CMake](#) as necessary to configure the project and generate build files for the main build tool.
 - Run the main build tool ([Ninja](#) or [GNU Make](#)). By default, the build tool is automatically detected but it can be explicitly set by passing the `-G` option to `idf.py`.

Building is incremental so if no source files or configuration has changed since the last build, nothing will be done.

- `idf.py clean` will “clean” the project by deleting build output files from the build directory, forcing a “full rebuild” the next time the project is built. Cleaning doesn’t delete CMake configuration output and some other files.
- `idf.py fullclean` will delete the entire “build” directory contents. This includes all CMake configuration output. The next time the project is built, CMake will configure it from scratch. Note that this option recursively deletes *all* files in the build directory, so use with care. Project configuration is not deleted.
- `idf.py flash` will automatically build the project if necessary, and then flash it to an ESP32. The `-p` and `-b` options can be used to set serial port name and flasher baud rate, respectively.
- `idf.py monitor` will display serial output from the ESP32. The `-p` option can be used to set the serial port name. Type `Ctrl-J` to exit the monitor. See [IDF Monitor](#) for more details about using the monitor.

Multiple `idf.py` commands can be combined into one. For example, `idf.py -p COM4 clean flash monitor` will clean the source tree, then build the project and flash it to the ESP32 before running the serial monitor.

For commands that are not known to `idf.py` an attempt to execute them as a build system target will be made.

Note: The environment variables `ESPPORT` and `ESPBAUD` can be used to set default values for the `-p` and `-b` options, respectively. Providing these options on the command line overrides the default.

Advanced Commands

- `idf.py app`, `idf.py bootloader`, `idf.py partition_table` can be used to build only the app, bootloader, or partition table from the project as applicable.
- There are matching commands `idf.py app-flash`, etc. to flash only that single part of the project to the ESP32.
- `idf.py -p PORT erase_flash` will use `esptool.py` to erase the ESP32’s entire flash chip.
- `idf.py size` prints some size information about the app. `size-components` and `size-files` are similar commands which print more detailed per-component or per-source-file information, respectively. If you define variable `-DOUTPUT_JSON=1` when running CMake (or `idf.py`), the output will be formatted as JSON not as human readable text.
- `idf.py reconfigure` re-runs [CMake](#) even if it doesn’t seem to need re-running. This isn’t necessary during normal usage, but can be useful after adding/removing files from the source tree, or when modifying CMake cache variables. For example, `idf.py -DNAME='VALUE' reconfigure` can be used to set variable `NAME` in CMake cache to value `VALUE`.
- `idf.py python-clean` deletes generated Python byte code from the IDF directory which may cause issues when switching between IDF and Python versions. It is advised to run this target after switching versions of Python.

The order of multiple `idf.py` commands on the same invocation is not important, they will automatically be executed in the correct order for everything to take effect (ie building before flashing, erasing before flashing, etc.).

idf.py options To list all available root level options, run `idf.py --help`. To list options that are specific for a subcommand, run `idf.py <command> --help`, for example `idf.py monitor --help`. Here is a list of some useful options:

- `-C <dir>` allows overriding the project directory from the default current working directory.
- `-B <dir>` allows overriding the build directory from the default `build` subdirectory of the project directory.

- `--ccache` flag can be used to enable [CCache](#) when compiling source files, if the [CCache](#) tool is installed. This can dramatically reduce some build times.

Note that some older versions of [CCache](#) may exhibit bugs on some platforms, so if files are not rebuilt as expected then try disabling [CCache](#) and build again. [CCache](#) can be enabled by default by setting the `IDF_CCACHE_ENABLE` environment variable to a non-zero value.

- `-v` flag causes both `idf.py` and the build system to produce verbose build output. This can be useful for debugging build problems.
- `--cmake-warn-uninitialized` (or `-w`) will cause [CMake](#) to print uninitialized variable warnings inside the project directory (not for directories not found inside the project directory). This only controls [CMake](#) variable warnings inside [CMake](#) itself, not other types of build warnings. This option can also be set permanently by setting the `IDF_CMAKE_WARN_UNINITIALIZED` environment variable to a non-zero value.

Using CMake Directly

`idf.py` is a wrapper around [CMake](#) for convenience. However, you can also invoke [CMake](#) directly if you prefer.

When `idf.py` does something, it prints each command that it runs for easy reference. For example, the `idf.py build` command is the same as running these commands in a bash shell (or similar commands for Windows Command Prompt):

```
mkdir -p build
cd build
cmake .. -G Ninja # or 'Unix Makefiles'
ninja
```

In the above list, the `cmake` command configures the project and generates build files for use with the final build tool. In this case the final build tool is [Ninja](#): running `ninja` actually builds the project.

It's not necessary to run `cmake` more than once. After the first build, you only need to run `ninja` each time. `ninja` will automatically re-invoke `cmake` if the project needs reconfiguration.

If using [CMake](#) with `ninja` or `make`, there are also targets for more of the `idf.py` sub-commands - for example running `make menuconfig` or `ninja menuconfig` in the build directory will work the same as `idf.py menuconfig`.

Note: If you're already familiar with [CMake](#), you may find the ESP-IDF [CMake](#)-based build system unusual because it wraps a lot of [CMake](#)'s functionality to reduce boilerplate. See [writing pure CMake components](#) for some information about writing more “[CMake style](#)” components.

Flashing with ninja or make It's possible to build and flash directly from `ninja` or `make` by running a target like:

```
ninja flash
```

Or:

```
make app-flash
```

Available targets are: `flash`, `app-flash` (app only), `bootloader-flash` (bootloader only).

When flashing this way, optionally set the `ESPPORT` and `ESPBAUD` environment variables to specify the serial port and baud rate. You can set environment variables in your operating system or IDE project. Alternatively, set them directly on the command line:

```
ESPPORT=/dev/ttyUSB0 ninja flash
```

Note: Providing environment variables at the start of the command like this is Bash shell Syntax. It will work on Linux and macOS. It won't work when using Windows Command Prompt, but it will work when using Bash-like shells on Windows.

Or:

```
make -j3 app-flash ESPPORT=COM4 ESPBAUD=2000000
```

Note: Providing variables at the end of the command line is make syntax, and works for make on all platforms.

Using CMake in an IDE

You can also use an IDE with CMake integration. The IDE will want to know the path to the project's `CMakeLists.txt` file. IDEs with CMake integration often provide their own build tools (CMake calls these "generators") to build the source files as part of the IDE.

When adding custom non-build steps like "flash" to the IDE, it is recommended to execute `idf.py` for these "special" commands.

For more detailed information about integrating ESP-IDF with CMake into an IDE, see [Build System Metadata](#).

Setting up the Python Interpreter

ESP-IDF works well with all supported Python versions. It should work out-of-box even if you have a legacy system where the default `python` interpreter is still Python 2.7, however, it is advised to switch to Python 3 if possible.

`idf.py` and other Python scripts will run with the default Python interpreter, i.e. `python`. You can switch to a different one like `python3 $IDF_PATH/tools/idf.py ...`, or you can set up a shell alias or another script to simplify the command.

If using CMake directly, running `cmake -D PYTHON=python3 ...` will cause CMake to override the default Python interpreter.

If using an IDE with CMake, setting the `PYTHON` value as a CMake cache override in the IDE UI will override the default Python interpreter.

To manage the Python version more generally via the command line, check out the tools [pyenv](#) or [virtualenv](#). These let you change the default Python version.

Possible issues The user of `idf.py` may sometimes experience `ImportError` described below.

```
Traceback (most recent call last):
  File "/Users/user_name/e/esp-idf/tools/kconfig_new/configgen.py", line 27, in
  ↳<module>
    import kconfiglib
ImportError: bad magic number in 'kconfiglib': b'\x03\xf3\r\n'
```

The exception is often caused by `.pyc` files generated by different Python versions. To solve the issue run the following command:

```
idf.py python-clean
```

5.4.3 Example Project

An example project directory tree might look like this:

```

- myProject/
  - CMakeLists.txt
  - sdkconfig
  - components/
    - component1/
      - CMakeLists.txt
      - Kconfig
      - src1.c
    - component2/
      - CMakeLists.txt
      - Kconfig
      - src1.c
      - include/
        - component2.h
  - main/
    - CMakeLists.txt
    - src1.c
    - src2.c
  - build/

```

This example “myProject” contains the following elements:

- A top-level project CMakeLists.txt file. This is the primary file which CMake uses to learn how to build the project; and may set project-wide CMake variables. It includes the file [/tools/cmake/project.cmake](#) which implements the rest of the build system. Finally, it sets the project name and defines the project.
- “sdkconfig” project configuration file. This file is created/updated when `idf.py menuconfig` runs, and holds configuration for all of the components in the project (including ESP-IDF itself). The “sdkconfig” file may or may not be added to the source control system of the project.
- Optional “components” directory contains components that are part of the project. A project does not have to contain custom components of this kind, but it can be useful for structuring reusable code or including third party components that aren’t part of ESP-IDF. Alternatively, `EXTRA_COMPONENT_DIRS` can be set in the top-level CMakeLists.txt to look for components in other places. See the [renaming main](#) section for more info. If you have a lot of source files in your project, we recommend grouping most into components instead of putting them all in “main” .
- “main” directory is a special component that contains source code for the project itself. “main” is a default name, the CMake variable `COMPONENT_DIRS` includes this component but you can modify this variable.
- “build” directory is where build output is created. This directory is created by `idf.py` if it doesn’t already exist. CMake configures the project and generates interim build files in this directory. Then, after the main build process is run, this directory will also contain interim object files and libraries as well as final binary output files. This directory is usually not added to source control or distributed with the project source code.

Component directories each contain a component CMakeLists.txt file. This file contains variable definitions to control the build process of the component, and its integration into the overall project. See [Component CMakeLists Files](#) for more details.

Each component may also include a Kconfig file defining the [component configuration](#) options that can be set via `menuconfig`. Some components may also include `Kconfig.projbuild` and `project_include.cmake` files, which are special files for [overriding parts of the project](#).

5.4.4 Project CMakeLists File

Each project has a single top-level CMakeLists.txt file that contains build settings for the entire project. By default, the project CMakeLists can be quite minimal.

Minimal Example CMakeLists

Minimal project:

```

cmake_minimum_required(VERSION 3.5)
include($ENV{IDF_PATH}/tools/cmake/project.cmake)
project(myProject)

```

Mandatory Parts

The inclusion of these three lines, in the order shown above, is necessary for every project:

- `cmake_minimum_required(VERSION 3.5)` tells CMake the minimum version that is required to build the project. ESP-IDF is designed to work with CMake 3.5 or newer. This line must be the first line in the `CMakeLists.txt` file.
- `include($ENV{IDF_PATH}/tools/cmake/project.cmake)` pulls in the rest of the CMake functionality to configure the project, discover all the components, etc.
- `project(myProject)` creates the project itself, and specifies the project name. The project name is used for the final binary output files of the app - ie `myProject.elf`, `myProject.bin`. Only one project can be defined per `CMakeLists` file.

Optional Project Variables

These variables all have default values that can be overridden for custom behaviour. Look in [/tools/cmake/project.cmake](#) for all of the implementation details.

- `COMPONENT_DIRS`, `COMPONENTS_DIRS`: Directories to search for components. Defaults to `IDF_PATH/components`, `PROJECT_DIR/components`, and `EXTRA_COMPONENT_DIRS`. Override this variable if you don't want to search for components in these places.
- `EXTRA_COMPONENT_DIRS`, `EXTRA_COMPONENTS_DIRS`: Optional list of additional directories to search for components. Paths can be relative to the project directory, or absolute.
- `COMPONENTS`: A list of component names to build into the project. Defaults to all components found in the `COMPONENT_DIRS` directories. Use this variable to “trim down” the project for faster build times. Note that any component which “requires” another component via the `REQUIRES` or `PRIV_REQUIRES` arguments on component registration will automatically have it added to this list, so the `COMPONENTS` list can be very short.

Any paths in these variables can be absolute paths, or set relative to the project directory.

To set these variables, use the `cmake set command` ie `set(VARIABLE "VALUE")`. The `set()` commands should be placed after the `cmake_minimum(...)` line but before the `include(...)` line.

Renaming main component

The build system provides special treatment to the `main` component. It is a component that gets automatically added to the build provided that it is in the expected location, `PROJECT_DIR/main`. All other components in the build are also added as its dependencies, saving the user from hunting down dependencies and providing a build that works right out of the box. Renaming the `main` component causes the loss of these behind-the-scenes heavy lifting, requiring the user to specify the location of the newly renamed component and manually specifying its dependencies. Specifically, the steps to renaming `main` are as follows:

1. Rename `main` directory.
2. Set `EXTRA_COMPONENT_DIRS` in the project `CMakeLists.txt` to include the renamed `main` directory.
3. Specify the dependencies in the renamed component's `CMakeLists.txt` file via `REQUIRES` or `PRIV_REQUIRES` arguments *on component registration*.

5.4.5 Component CMakeLists Files

Each project contains one or more components. Components can be part of ESP-IDF, part of the project's own components directory, or added from custom component directories (*see above*).

A component is any directory in the `COMPONENT_DIRS` list which contains a `CMakeLists.txt` file.

Searching for Components

The list of directories in `COMPONENT_DIRS` is searched for the project's components. Directories in this list can either be components themselves (ie they contain a `CMakeLists.txt` file), or they can be top-level directories whose sub-directories are components.

When CMake runs to configure the project, it logs the components included in the build. This list can be useful for debugging the inclusion/exclusion of certain components.

Multiple components with the same name

When ESP-IDF is collecting all the components to compile, it will do this in the order specified by `COMPONENT_DIRS`; by default, this means ESP-IDF's internal components first, then the project's components, and finally any components set in `EXTRA_COMPONENT_DIRS`. If two or more of these directories contain component sub-directories with the same name, the component in the last place searched is used. This allows, for example, overriding ESP-IDF components with a modified version by copying that component from the ESP-IDF components directory to the project components directory and then modifying it there. If used in this way, the ESP-IDF directory itself can remain untouched.

Minimal Component CMakeLists

The minimal component `CMakeLists.txt` file simply registers the component to the build system using `idf_component_register`:

```
idf_component_register(SRCS "foo.c" "bar.c"
                      INCLUDE_DIRS "include"
                      REQUIRES mbedtls)
```

- `SRCS` is a list of source files (`*.c`, `*.cpp`, `*.cc`, `*.S`). These source files will be compiled into the component library.
- `INCLUDE_DIRS` is a list of directories to add to the global include search path for any component which requires this component, and also the main source files.
- `REQUIRES` is not actually required, but it is very often required to declare what other components this component will use. See [Component Requirements](#).

A library with the name of the component will be built and linked into the final app.

Directories are usually specified relative to the `CMakeLists.txt` file itself, although they can be absolute.

There are other arguments that can be passed to `idf_component_register`. These arguments are discussed [here](#).

See [example component requirements](#) and [example component CMakeLists](#) for more complete component `CMakeLists.txt` examples.

Preset Component Variables

The following component-specific variables are available for use inside component `CMakeLists`, but should not be modified:

- `COMPONENT_DIR`: The component directory. Evaluates to the absolute path of the directory containing `CMakeLists.txt`. The component path cannot contain spaces. This is the same as the `CMAKE_CURRENT_SOURCE_DIR` variable.
- `COMPONENT_NAME`: Name of the component. Same as the name of the component directory.
- `COMPONENT_ALIAS`: Alias of the library created internally by the build system for the component.
- `COMPONENT_LIB`: Name of the library created internally by the build system for the component.

The following variables are set at the project level, but available for use in component `CMakeLists`:

- `CONFIG_*`: Each value in the project configuration has a corresponding variable available in cmake. All names begin with `CONFIG_`. [More information here](#).
- `ESP_PLATFORM`: Set to 1 when the CMake file is processed within ESP-IDF build system.

Build/Project Variables

The following are some project/build variables that are available as build properties and whose values can be queried using `idf_build_get_property` from the component `CMakeLists.txt`:

- `PROJECT_NAME`: Name of the project, as set in project `CMakeLists.txt` file.
- `PROJECT_DIR`: Absolute path of the project directory containing the project `CMakeLists`. Same as the `CMAKE_SOURCE_DIR` variable.
- `COMPONENTS`: Names of all components that are included in this build, formatted as a semicolon-delimited CMake list.
- `IDF_VER`: Git version of ESP-IDF (produced by `git describe`)
- `IDF_VERSION_MAJOR`, `IDF_VERSION_MINOR`, `IDF_VERSION_PATCH`: Components of ESP-IDF version, to be used in conditional expressions. Note that this information is less precise than that provided by `IDF_VER` variable. `v4.0-dev-*`, `v4.0-beta1`, `v4.0-rc1` and `v4.0` will all have the same values of `IDF_VERSION_*` variables, but different `IDF_VER` values.
- `IDF_TARGET`: Name of the target for which the project is being built.
- `PROJECT_VER`: Project version.
 - If `PROJECT_VER` variable is set in project `CMakeLists.txt` file, its value will be used.
 - Else, if the `PROJECT_DIR/version.txt` exists, its contents will be used as `PROJECT_VER`.
 - Else, if the project is located inside a Git repository, the output of `git describe` will be used.
 - Otherwise, `PROJECT_VER` will be “1” .

Other build properties are listed [here](#).

Controlling Component Compilation

To pass compiler options when compiling source files belonging to a particular component, use the `target_compile_options` function:

```
target_compile_options(${COMPONENT_LIB} PRIVATE -Wno-unused-variable)
```

To apply the compilation flags to a single source file, use the CMake `set_source_files_properties` command:

```
set_source_files_properties(mysrc.c
    PROPERTIES COMPILE_FLAGS
        -Wno-unused-variable
)
```

This can be useful if there is upstream code that emits warnings.

When using these commands, place them after the call to `idf_component_register` in the component `CMakeLists` file.

5.4.6 Component Configuration

Each component can also have a `Kconfig` file, alongside `CMakeLists.txt`. This contains configuration settings to add to the configuration menu for this component.

These settings are found under the “Component Settings” menu when `menuconfig` is run.

To create a component `Kconfig` file, it is easiest to start with one of the `Kconfig` files distributed with ESP-IDF.

For an example, see [Adding conditional configuration](#).

5.4.7 Preprocessor Definitions

The ESP-IDF build system adds the following C preprocessor definitions on the command line:

- `ESP_PLATFORM`: Can be used to detect that build happens within ESP-IDF.
- `IDF_VER`: Defined to a git version string. E.g. `v2.0` for a tagged release or `v1.0-275-g0efaa4f` for an arbitrary commit.

5.4.8 Component Requirements

When compiling each component, the ESP-IDF build system recursively evaluates its dependencies. This means each component needs to declare the components that it depends on (“requires”).

When writing a component

```
idf_component_register(...
    REQUIRES mbedtls
    PRIV_REQUIRES console spiffs)
```

- `REQUIRES` should be set to all components whose header files are `#included` from the *public* header files of this component.
- `PRIV_REQUIRES` should be set to all components whose header files are `#included` from *any source files* in this component, unless already listed in `REQUIRES`. Also any component which is required to be linked in order for this component to function correctly.
- The values of `REQUIRES` and `PRIV_REQUIRES` should not depend on any configuration choices (`CONFIG_XXX` macros). This is because requirements are expanded before configuration is loaded. Other component variables (like include paths or source files) can depend on configuration choices.
- Not setting either or both `REQUIRES` variables is fine. If the component has no requirements except for the *Common component requirements* needed for RTOS, libc, etc.

If a components only supports some target chips (values of `IDF_TARGET`) then it can specify `REQUIRED_IDF_TARGETS` in the `idf_component_register` call to express these requirements. In this case the build system will generate an error if the component is included into the build, but does not support the selected target.

Note: In CMake terms, `REQUIRES` & `PRIV_REQUIRES` are approximate wrappers around the CMake functions `target_link_libraries(... PUBLIC ...)` and `target_link_libraries(... PRIVATE ...)`.

Example of component requirements

Imagine there is a `car` component, which uses the `engine` component, which uses the `spark_plug` component:

```
- autoProject/
  - CMakeLists.txt
  - components/ - car/ - CMakeLists.txt
                    - car.c
                    - car.h
  - engine/ - CMakeLists.txt
            - engine.c
            - include/ - engine.h
  - spark_plug/ - CMakeLists.txt
                - plug.c
                - plug.h
```


Car component The `car.h` header file is the public interface for the `car` component. This header includes `engine.h` directly because it uses some declarations from this header:

```
/* car.h */
#include "engine.h"

#ifdef ENGINE_IS_HYBRID
#define CAR_MODEL "Hybrid"
#endif
```

And `car.c` includes `car.h` as well:

```
/* car.c */
#include "car.h"
```

This means the `car/CMakeLists.txt` file needs to declare that `car` requires `engine`:

```
idf_component_register(SRCS "car.c"
                      INCLUDE_DIRS "."
                      REQUIRES engine)
```

- `SRCS` gives the list of source files in the `car` component.
- `INCLUDE_DIRS` gives the list of public include directories for this component. Because the public interface is `car.h`, the directory containing `car.h` is listed here.
- `REQUIRES` gives the list of components required by the public interface of this component. Because `car.h` is a public header and includes a header from `engine`, we include `engine` here. This makes sure that any other component which includes `car.h` will be able to recursively include the required `engine.h` also.

Engine component The `engine` component also has a public header file `include/engine.h`, but this header is simpler:

```
/* engine.h */
#define ENGINE_IS_HYBRID

void engine_start(void);
```

The implementation is in `engine.c`:

```
/* engine.c */
#include "engine.h"
#include "spark_plug.h"

...
```

In this component, `engine` depends on `spark_plug` but this is a private dependency. `spark_plug.h` is needed to compile `engine.c`, but not needed to include `engine.h`.

This means that the `engine/CMakeLists.txt` file can use `PRIV_REQUIRES`:

```
idf_component_register(SRCS "engine.c"
                      INCLUDE_DIRS "include"
                      PRIV_REQUIRES spark_plug)
```

As a result, source files in the `car` component don't need the `spark_plug` include directories added to their compiler search path. This can speed up compilation, and stops compiler command lines from becoming longer than necessary.

Spark Plug Component The `spark_plug` component doesn't depend on anything else. It has a public header file `spark_plug.h`, but this doesn't include headers from any other components.

This means that the `spark_plug/CMakeLists.txt` file doesn't need any `REQUIRES` or `PRIV_REQUIRES` clauses:

```
idf_component_register(SRCS "spark_plug.c"
                      INCLUDE_DIRS ".")
```

Source File Include Directories

Each component's source file is compiled with these include path directories, as specified in the passed arguments to `idf_component_register`:

```
idf_component_register(..
                      INCLUDE_DIRS "include"
                      PRIV_INCLUDE_DIRS "other")
```

- The current component's `INCLUDE_DIRS` and `PRIV_INCLUDE_DIRS`.
- The `INCLUDE_DIRS` belonging to all other components listed in the `REQUIRES` and `PRIV_REQUIRES` parameters (ie all the current component's public and private dependencies).
- Recursively, all of the `INCLUDE_DIRS` of those components `REQUIRES` lists (ie all public dependencies of this component's dependencies, recursively expanded).

Main component requirements

The component named `main` is special because it automatically requires all other components in the build. So it's not necessary to pass `REQUIRES` or `PRIV_REQUIRES` to this component. See [renaming main](#) for a description of what needs to be changed if no longer using the `main` component.

Common component requirements

To avoid duplication, every component automatically requires some “common” IDF components even if they are not mentioned explicitly. Headers from these components can always be included.

The list of common components is: `freertos`, `newlib`, `heap`, `log`, `soc`, `esp_rom`, `esp_common`, `xtensa`, `cxx`.

Including components in the build

- By default, every component is included in the build.
- If you set the `COMPONENTS` variable to a minimal list of components used directly by your project, then the build will expand to also include required components. The full list of components will be:
 - Components mentioned explicitly in `COMPONENTS`.
 - Those components' requirements (evaluated recursively).
 - The “common” components that every component depends on.
- Setting `COMPONENTS` to the minimal list of required components can significantly reduce compile times.

Requirements in the build system implementation

- Very early in the CMake configuration process, the script `expand_requirements.cmake` is run. This script does a partial evaluation of all component `CMakeLists.txt` files and builds a graph of component requirements (this graph may have cycles). The graph is used to generate a file `component_depends.cmake` in the build directory.
- The main CMake process then includes this file and uses it to determine the list of components to include in the build (internal `BUILD_COMPONENTS` variable). The `BUILD_COMPONENTS` variable is sorted so dependencies are listed first, however as the component dependency graph has cycles this cannot be guaranteed for all components. The order should be deterministic given the same set of components and component dependencies.

- The value of `BUILD_COMPONENTS` is logged by CMake as “Component names: “
- Configuration is then evaluated for the components included in the build.
- Each component is included in the build normally and the `CMakeLists.txt` file is evaluated again to add the component libraries to the build.

Component Dependency Order The order of components in the `BUILD_COMPONENTS` variable determines other orderings during the build:

- Order that `project_include.cmake` files are included into the project.
- Order that the list of header paths is generated for compilation (via `-I` argument). (Note that for a given component’s source files, only that component’s dependency’s header paths are passed to the compiler.)

5.4.9 Overriding Parts of the Project

`project_include.cmake`

For components that have build requirements which must be evaluated before any component `CMakeLists` files are evaluated, you can create a file called `project_include.cmake` in the component directory. This CMake file is included when `project.cmake` is evaluating the entire project.

`project_include.cmake` files are used inside ESP-IDF, for defining project-wide build features such as `esptool.py` command line arguments and the bootloader “special app” .

Unlike component `CMakeLists.txt` files, when including a `project_include.cmake` file the current source directory (`CMAKE_CURRENT_SOURCE_DIR` and working directory) is the project directory. Use the variable `COMPONENT_DIR` for the absolute directory of the component.

Note that `project_include.cmake` isn’t necessary for the most common component uses - such as adding include directories to the project, or `LDFLAGS` to the final linking step. These values can be customised via the `CMakeLists.txt` file itself. See *Optional Project Variables* for details.

`project_include.cmake` files are included in the order given in `BUILD_COMPONENTS` variable (as logged by CMake). This means that a component’s `project_include.cmake` file will be included after it’s all dependencies’ `project_include.cmake` files, unless both components are part of a dependency cycle. This is important if a `project_include.cmake` file relies on variables set by another component. See also *above*.

Take great care when setting variables or targets in a `project_include.cmake` file. As the values are included into the top-level project CMake pass, they can influence or break functionality across all components!

`KConfig.projbuild`

This is an equivalent to `project_include.cmake` for *Component Configuration* `KConfig` files. If you want to include configuration options at the top-level of `menuconfig`, rather than inside the “Component Configuration” sub-menu, then these can be defined in the `KConfig.projbuild` file alongside the `CMakeLists.txt` file.

Take care when adding configuration values in this file, as they will be included across the entire project configuration. Where possible, it’s generally better to create a `KConfig` file for *Component Configuration*.

`project_include.cmake` files are used inside ESP-IDF, for defining project-wide build features such as `esptool.py` command line arguments and the bootloader “special app” .

5.4.10 Configuration-Only Components

Special components which contain no source files, only `Kconfig.projbuild` and `KConfig`, can have a one-line `CMakeLists.txt` file which calls the function `idf_component_register()` with no arguments specified. This function will include the component in the project build, but no library will be built *and* no header files will be added to any include paths.

5.4.11 Debugging CMake

For full details about CMake and CMake commands, see the [CMake v3.5 documentation](#).

Some tips for debugging the ESP-IDF CMake-based build system:

- When CMake runs, it prints quite a lot of diagnostic information including lists of components and component paths.
- Running `cmake -DDEBUG=1` will produce more verbose diagnostic output from the IDF build system.
- Running `cmake` with the `--trace` or `--trace-expand` options will give a lot of information about control flow. See the [cmake command line documentation](#).

When included from a project CMakeLists file, the `project.cmake` file defines some utility modules and global variables and then sets `IDF_PATH` if it was not set in the system environment.

It also defines an overridden custom version of the built-in CMake `project` function. This function is overridden to add all of the ESP-IDF specific project functionality.

Warning On Undefined Variables

By default, `idf.py` passes the `--warn-uninitialized` flag to CMake so it will print a warning if an undefined variable is referenced in the build. This can be very useful to find buggy CMake files.

If you don't want this behaviour, it can be disabled by passing `--no-warnings` to `idf.py`.

Browse the [/tools/cmake/project.cmake](#) file and supporting functions in [/tools/cmake/idf_functions.cmake](#) for more details.

5.4.12 Example Component CMakeLists

Because the build environment tries to set reasonable defaults that will work most of the time, component `CMakeLists.txt` can be very small or even empty (see [Minimal Component CMakeLists](#)). However, overriding *component variables* is usually required for some functionality.

Here are some more advanced examples of component CMakeLists files.

Adding conditional configuration

The configuration system can be used to conditionally compile some files depending on the options selected in the project configuration.

Kconfig:

```
config FOO_ENABLE_BAR
    bool "Enable the BAR feature."
    help
        This enables the BAR feature of the FOO component.
```

CMakeLists.txt:

```
set(srcs "foo.c" "more_foo.c")

if(CONFIG_FOO_ENABLE_BAR)
    list(APPEND srcs "bar.c")
endif()

idf_component_register(SRCS "${srcs}"
    ...)
```

This example makes use of the CMake `if` function and `list APPEND` function.

This can also be used to select or stub out an implementation, as such:

Kconfig:

```
config ENABLE_LCD_OUTPUT
    bool "Enable LCD output."
    help
        Select this if your board has a LCD.

config ENABLE_LCD_CONSOLE
    bool "Output console text to LCD"
    depends on ENABLE_LCD_OUTPUT
    help
        Select this to output debugging output to the lcd

config ENABLE_LCD_PLOT
    bool "Output temperature plots to LCD"
    depends on ENABLE_LCD_OUTPUT
    help
        Select this to output temperature plots
```

CMakeLists.txt:

```
if(CONFIG_ENABLE_LCD_OUTPUT)
    set(srcs lcd-real.c lcd-spi.c)
else()
    set(srcs lcd-dummy.c)
endif()

# We need font if either console or plot is enabled
if(CONFIG_ENABLE_LCD_CONSOLE OR CONFIG_ENABLE_LCD_PLOT)
    list(APPEND srcs "font.c")
endif()

idf_component_register(SRCS "${srcs}"
    ...)
```

Conditions which depend on the target

The current target is available to CMake files via `IDF_TARGET` variable.

In addition to that, if target `xyz` is used (`IDF_TARGET=xyz`), then Kconfig variable `CONFIG_IDF_TARGET_XYZ` will be set.

Note that component dependencies may depend on `IDF_TARGET` variable, but not on Kconfig variables. Also one can not use Kconfig variables in `include` statements in CMake files, but `IDF_TARGET` can be used in such context.

Source Code Generation

Some components will have a situation where a source file isn't supplied with the component itself but has to be generated from another file. Say our component has a header file that consists of the converted binary data of a BMP file, converted using a hypothetical tool called `bmp2h`. The header file is then included in as C source file called `graphics_lib.c`:

```
add_custom_command(OUTPUT logo.h
    COMMAND bmp2h -i ${COMPONENT_DIR}/logo.bmp -o log.h
    DEPENDS ${COMPONENT_DIR}/logo.bmp
    VERBATIM)
```

(continues on next page)

(continued from previous page)

```
add_custom_target(logo DEPENDS logo.h)
add_dependencies(${COMPONENT_LIB} logo)

set_property(DIRECTORY "${COMPONENT_DIR}" APPEND PROPERTY
             ADDITIONAL_MAKE_CLEAN_FILES logo.h)
```

This answer is adapted from the [CMake FAQ entry](#), which contains some other examples that will also work with ESP-IDF builds.

In this example, `logo.h` will be generated in the current directory (the build directory) while `logo.bmp` comes with the component and resides under the component path. Because `logo.h` is a generated file, it should be cleaned when the project is cleaned. For this reason it is added to the `ADDITIONAL_MAKE_CLEAN_FILES` property.

Note: If generating files as part of the project `CMakeLists.txt` file, not a component `CMakeLists.txt`, then use build property `PROJECT_DIR` instead of `${COMPONENT_DIR}` and `${PROJECT_NAME}.elf` instead of `${COMPONENT_LIB}`.)

If a source file from another component included `logo.h`, then `add_dependencies` would need to be called to add a dependency between the two components, to ensure that the component source files were always compiled in the correct order.

Embedding Binary Data

Sometimes you have a file with some binary or text data that you'd like to make available to your component - but you don't want to reformat the file as C source.

You can specify argument `EMBED_FILES` in the component registration, giving space-delimited names of the files to embed:

```
idf_component_register(...
                      EMBED_FILES server_root_cert.der)
```

Or if the file is a string, you can use the variable `EMBED_TXTFILES`. This will embed the contents of the text file as a null-terminated string:

```
idf_component_register(...
                      EMBED_TXTFILES server_root_cert.pem)
```

The file's contents will be added to the `.rodata` section in flash, and are available via symbol names as follows:

```
extern const uint8_t server_root_cert_pem_start[] asm("_binary_server_root_cert_
↪pem_start");
extern const uint8_t server_root_cert_pem_end[]   asm("_binary_server_root_cert_
↪pem_end");
```

The names are generated from the full name of the file, as given in `EMBED_FILES`. Characters `/`, `.`, etc. are replaced with underscores. The `_binary` prefix in the symbol name is added by `objcopy` and is the same for both text and binary files.

To embed a file into a project, rather than a component, you can call the function `target_add_binary_data` like this:

```
target_add_binary_data(myproject.elf "main/data.bin" TEXT)
```

Place this line after the `project()` line in your project `CMakeLists.txt` file. Replace `myproject.elf` with your project name. The final argument can be `TEXT` to embed a null-terminated string, or `BINARY` to embed the content as-is.

For an example of using this technique, see [protocols/https_request](#) - the certificate file contents are loaded from the text .pem file at compile time.

Code and Data Placements

ESP-IDF has a feature called linker script generation that enables components to define where its code and data will be placed in memory through linker fragment files. These files are processed by the build system, and is used to augment the linker script used for linking app binary. See [Linker Script Generation](#) for a quick start guide as well as a detailed discussion of the mechanism.

Fully Overriding The Component Build Process

Obviously, there are cases where all these recipes are insufficient for a certain component, for example when the component is basically a wrapper around another third-party component not originally intended to be compiled under this build system. In that case, it's possible to forego the ESP-IDF build system entirely by using a CMake feature called [ExternalProject](#). Example component CMakeLists:

```
# External build process for quirc, runs in source dir and
# produces libquirc.a
externalproject_add(quirc_build
  PREFIX ${COMPONENT_DIR}
  SOURCE_DIR ${COMPONENT_DIR}/quirc
  CONFIGURE_COMMAND ""
  BUILD_IN_SOURCE 1
  BUILD_COMMAND make CC=${CMAKE_C_COMPILER} libquirc.a
  INSTALL_COMMAND ""
)

# Add libquirc.a to the build process
add_library(quirc STATIC IMPORTED GLOBAL)
add_dependencies(quirc quirc_build)

set_target_properties(quirc PROPERTIES IMPORTED_LOCATION
  ${COMPONENT_DIR}/quirc/libquirc.a)
set_target_properties(quirc PROPERTIES INTERFACE_INCLUDE_DIRECTORIES
  ${COMPONENT_DIR}/quirc/lib)

set_directory_properties( PROPERTIES ADDITIONAL_MAKE_CLEAN_FILES
  "${COMPONENT_DIR}/quirc/libquirc.a")
```

(The above CMakeLists.txt can be used to create a component named `quirc` that builds the `quirc` project using its own Makefile.)

- `externalproject_add` defines an external build system.
 - `SOURCE_DIR`, `CONFIGURE_COMMAND`, `BUILD_COMMAND` and `INSTALL_COMMAND` should always be set. `CONFIGURE_COMMAND` can be set to an empty string if the build system has no “configure” step. `INSTALL_COMMAND` will generally be empty for ESP-IDF builds.
 - Setting `BUILD_IN_SOURCE` means the build directory is the same as the source directory. Otherwise you can set `BUILD_DIR`.
 - Consult the [ExternalProject](#) documentation for more details about `externalproject_add()`
- The second set of commands adds a library target, which points to the “imported” library file built by the external system. Some properties need to be set in order to add include directories and tell CMake where this file is.
- Finally, the generated library is added to `ADDITIONAL_MAKE_CLEAN_FILES`. This means `make clean` will delete this library. (Note that the other object files from the build won't be deleted.)

Note: When using an external build process with PSRAM, remember to add `-mfix-esp32-psram-cache-issue` to the C compiler arguments. See [CONFIG_SPIRAM_CACHE_WORKAROUND](#) for details of this flag.

ExternalProject dependencies, clean builds CMake has some unusual behaviour around external project builds:

- `ADDITIONAL_MAKE_CLEAN_FILES` only works when “make” is used as the build system. If Ninja or an IDE build system is used, it won't delete these files when cleaning.
- However, the `ExternalProject` configure & build commands will *always* be re-run after a clean is run.
- Therefore, there are two alternative recommended ways to configure the external build command:
 1. Have the external `BUILD_COMMAND` run a full clean compile of all sources. The build command will be run if any of the dependencies passed to `externalproject_add` with `DEPENDS` have changed, or if this is a clean build (ie any of `idf.py clean`, `ninja clean`, or `make clean` was run.)
 2. Have the external `BUILD_COMMAND` be an incremental build command. Pass the parameter `BUILD_ALWAYS 1` to `externalproject_add`. This means the external project will be built each time a build is run, regardless of dependencies. This is only recommended if the external project has correct incremental build behaviour, and doesn't take too long to run.

The best of these approaches for building an external project will depend on the project itself, its build system, and whether you anticipate needing to frequently recompile the project.

5.4.13 Custom sdkconfig defaults

For example projects or other projects where you don't want to specify a full `sdkconfig` configuration, but you do want to override some key values from the ESP-IDF defaults, it is possible to create a file `sdkconfig.defaults` in the project directory. This file will be used when creating a new config from scratch, or when any new config value hasn't yet been set in the `sdkconfig` file.

To override the name of this file or to specify multiple files, set the `SDKCONFIG_DEFAULTS` environment variable or set `SDKCONFIG_DEFAULTS` in top-level `CMakeLists.txt`. If specifying multiple files, use semicolon as the list separator. File names not specified as full paths are resolved relative to current project.

Target-dependent sdkconfig defaults

In addition to `sdkconfig.defaults` file, build system will also load defaults from `sdkconfig.defaults.TARGET_NAME` file, where `TARGET_NAME` is the value of `IDF_TARGET`. For example, for `esp32` target, default settings will be taken from `sdkconfig.defaults` first, and then from `sdkconfig.defaults.esp32`.

If `SDKCONFIG_DEFAULTS` is used to override the name of defaults file/files, the name of target-specific defaults file will be derived from `SDKCONFIG_DEFAULTS` value/values using the rule above.

5.4.14 Flash arguments

There are some scenarios that we want to flash the target board without IDF. For this case we want to save the built binaries, `esptool.py` and `esptool write_flash` arguments. It's simple to write a script to save binaries and `esptool.py`.

After running a project build, the build directory contains binary output files (`.bin` files) for the project and also the following flashing data files:

- `flash_project_args` contains arguments to flash the entire project (app, bootloader, partition table, PHY data if this is configured).
- `flash_app_args` contains arguments to flash only the app.
- `flash_bootloader_args` contains arguments to flash only the bootloader.

You can pass any of these flasher argument files to `esptool.py` as follows:

```
python esptool.py --chip esp32 write_flash @build/flash_project_args
```

Alternatively, it is possible to manually copy the parameters from the argument file and pass them on the command line.

The build directory also contains a generated file `flasher_args.json` which contains project flash information, in JSON format. This file is used by `idf.py` and can also be used by other tools which need information about the project build.

5.4.15 Building the Bootloader

The bootloader is built by default as part of `idf.py build`, or can be built standalone via `idf.py bootloader`.

The bootloader is a special “subproject” inside `/components/bootloader/subproject`. It has its own project CMakeLists.txt file and builds separate .ELF and .BIN files to the main project. However it shares its configuration and build directory with the main project.

The subproject is inserted as an external project from the top-level project, by the file `/components/bootloader/project_include.cmake`. The main build process runs CMake for the subproject, which includes discovering components (a subset of the main components) and generating a bootloader-specific config (derived from the main `sdkconfig`).

5.4.16 Selecting the Target

ESP-IDF supports multiple targets (chips). The identifiers used for each chip are as follows:

- `esp32` —for ESP32-D0WD, ESP32-D2WD, ESP32-S0WD (ESP-SOLO), ESP32-U4WDH, ESP32-PICO-D4
- `esp32s2beta`—for ESP32-S2-beta (engineering samples)

To select the target before building the project, use `idf.py set-target <target>` command, for example:

```
idf.py set-target esp32s2beta
```

Important: `idf.py set-target` will clear the build directory and re-generate the `sdkconfig` file from scratch. The old `sdkconfig` file will be saved as `sdkconfig.old`.

Note: The behavior of `idf.py set-target` command is equivalent to:

1. clearing the build directory (`idf.py fullclean`)
 2. removing the `sdkconfig` file (`mv sdkconfig sdkconfig.old`)
 3. configuring the project with the new target (`idf.py -DIDF_TARGET=esp32 reconfigure`)
-

It is also possible to pass the desired `IDF_TARGET` as an environment variable (e.g. `export IDF_TARGET=esp32s2beta`) or as a CMake variable (e.g. `-DIDF_TARGET=esp32s2beta` argument to CMake or `idf.py`). Setting the environment variable is a convenient method if you mostly work with one type of the chip.

To specify the `_default_` value of `IDF_TARGET` for a given project, add `CONFIG_IDF_TARGET` value to `sdkconfig.defaults`. For example, `CONFIG_IDF_TARGET="esp32s2beta"`. This value will be used if `IDF_TARGET` is not specified by other method: using an environment variable, CMake variable, or `idf.py set-target` command.

If the target has not been set by any of these methods, the build system will default to `esp32` target.

5.4.17 Writing Pure CMake Components

The ESP-IDF build system “wraps” CMake with the concept of “components”, and helper functions to automatically integrate these components into a project build.

However, underneath the concept of “components” is a full CMake build system. It is also possible to make a component which is pure CMake.

Here is an example minimal “pure CMake” component CMakeLists file for a component named `json`:

```
add_library(json STATIC
cJSON/cJSON.c
cJSON/cJSON_Utils.c)

target_include_directories(json PUBLIC cJSON)
```

- This is actually an equivalent declaration to the IDF `json` component `/components/json/CMakeLists.txt`.
- This file is quite simple as there are not a lot of source files. For components with a large number of files, the globbing behaviour of ESP-IDF's component logic can make the component `CMakeLists` style simpler.)
- Any time a component adds a library target with the component name, the ESP-IDF build system will automatically add this to the build, expose public include directories, etc. If a component wants to add a library target with a different name, dependencies will need to be added manually via CMake commands.

5.4.18 Using Third-Party CMake Projects with Components

CMake is used for a lot of open-source C and C++ projects —code that users can tap into for their applications. One of the benefits of having a CMake build system is the ability to import these third-party projects, sometimes even without modification! This allows for users to be able to get functionality that may not yet be provided by a component, or use another library for the same functionality.

Importing a library might look like this for a hypothetical library `foo` to be used in the `main` component:

```
# Register the component
idf_component_register(...)

# Set values of hypothetical variables that control the build of `foo`
set(FOO_BUILD_STATIC OFF)
set(FOO_BUILD_TESTS OFF)

# Create and import the library targets
add_subdirectory(foo)

# Publicly link `foo` to `main` component
target_link_libraries(main PUBLIC foo)
```

For an actual example, take a look at `build_system/cmake/import_lib`. Take note that what needs to be done in order to import the library may vary. It is recommended to read up on the library's documentation for instructions on how to import it from other projects. Studying the library's `CMakeLists.txt` and build structure can also be helpful.

It is also possible to wrap a third-party library to be used as a component in this manner. For example, the `mbedtls` component is a wrapper for Espressif's fork of `mbedtls`. See its `component CMakeLists.txt`.

The CMake variable `ESP_PLATFORM` is set to 1 whenever the ESP-IDF build system is being used. Tests such as `if (ESP_PLATFORM)` can be used in generic CMake code if special IDF-specific logic is required.

Using ESP-IDF components from external libraries

The above example assumes that the external library `foo` (or `tinyclib` in the case of the `import_lib` example) doesn't need to use any ESP-IDF APIs apart from common APIs such as `libc`, `libstdc++`, etc. If the external library needs to use APIs provided by other ESP-IDF components, this needs to be specified in the external `CMakeLists.txt` file by adding a dependency on the library target `idf::<componentname>`.

For example, in the `foo/CMakeLists.txt` file:

```
add_library(foo bar.c fizz.cpp buzz.cpp)

if(ESP_PLATFORM)
  # On ESP-IDF, bar.c needs to include esp_spi_flash.h from the spi_flash component
  target_link_libraries(foo PRIVATE idf::spi_flash)
endif()
```

5.4.19 Using Prebuilt Libraries with Components

Another possibility is that you have a prebuilt static library (.a file), built by some other build process.

The ESP-IDF build system provides a utility function `add_prebuilt_library` for users to be able to easily import and use prebuilt libraries:

```
add_prebuilt_library(target_name lib_path [REQUIRES req1 req2 ...] [PRIV_REQUIRES_
↳req1 req2 ...])
```

where:

- `target_name`- name that can be used to reference the imported library, such as when linking to other targets
- `lib_path`- path to prebuilt library; may be an absolute or relative path to the component directory

Optional arguments `REQUIRES` and `PRIV_REQUIRES` specify dependency on other components. These have the same meaning as the arguments for `idf_component_register`.

Take note that the prebuilt library must have been compiled for the same target as the consuming project. Configuration relevant to the prebuilt library must also match. If not paid attention to, these two factors may contribute to subtle bugs in the app.

For an example, take a look at [build_system/cmake/import_prebuilt](#).

5.4.20 Using ESP-IDF in Custom CMake Projects

ESP-IDF provides a template CMake project for easily creating an application. However, in some instances the user might already have an existing CMake project or may want to create a custom one. In these cases it is desirable to be able to consume IDF components as libraries to be linked to the user's targets (libraries/ executables).

It is possible to do so by using the *build system APIs provided* by `tools/cmake/idf.cmake`. For example:

```
cmake_minimum_required(VERSION 3.5)
project(my_custom_app C)

# Include CMake file that provides ESP-IDF CMake build system APIs.
include($ENV{IDF_PATH}/tools/cmake/idf.cmake)

# Include ESP-IDF components in the build, may be thought as an equivalent of
# add_subdirectory() but with some additional processing and magic for ESP-IDF_
↳build
# specific build processes.
idf_build_process(esp32)

# Create the project executable and plainly link the newlib component to it using
# its alias, idf::newlib.
add_executable(${CMAKE_PROJECT_NAME}.elf main.c)
target_link_libraries(${CMAKE_PROJECT_NAME}.elf idf::newlib)

# Let the build system know what the project executable is to attach more targets,
↳dependencies, etc.
idf_build_executable(${CMAKE_PROJECT_NAME}.elf)
```

The example in [build_system/cmake/idf_as_lib](#) demonstrates the creation of an application equivalent to [hello world application](#) using a custom CMake project.

Note: The IDF build system can only set compiler flags for source files that it builds. When an external CMake-Lists.txt file is used and PSRAM is enabled, remember to add `-mfix-esp32-psram-cache-issue` to the C compiler arguments. See [CONFIG_SPIRAM_CACHE_WORKAROUND](#) for details of this flag.

5.4.21 ESP-IDF CMake Build System API

idf-build-commands

```
idf_build_get_property(var property [GENERATOR_EXPRESSION])
```

Retrieve a *build property* *property* and store it in *var* accessible from the current scope. Specifying *GENERATOR_EXPRESSION* will retrieve the generator expression string for that property, instead of the actual value, which can be used with CMake commands that support generator expressions.

```
idf_build_set_property(property val [APPEND])
```

Set a *build property* *property* with value *val*. Specifying *APPEND* will append the specified value to the current value of the property. If the property does not previously exist or it is currently empty, the specified value becomes the first element/member instead.

```
idf_build_component(component_dir)
```

Present a directory *component_dir* that contains a component to the build system. Relative paths are converted to absolute paths with respect to current directory. All calls to this command must be performed before *idf_build_process*.

This command does not guarantee that the component will be processed during build (see the *COMPONENTS* argument description for *idf_build_process*)

```
idf_build_process(target
    [PROJECT_DIR project_dir]
    [PROJECT_VER project_ver]
    [PROJECT_NAME project_name]
    [SDKCONFIG sdkconfig]
    [SDKCONFIG_DEFAULTS sdkconfig_defaults]
    [BUILD_DIR build_dir]
    [COMPONENTS component1 component2 ...])
```

Performs the bulk of the behind-the-scenes magic for including ESP-IDF components such as component configuration, libraries creation, dependency expansion and resolution. Among these functions, perhaps the most important from a user's perspective is the libraries creation by calling each component's *idf_component_register*. This command creates the libraries for each component, which are accessible using aliases in the form *idf::component_name*. These aliases can be used to link the components to the user's own targets, either libraries or executables.

The call requires the target chip to be specified with *target* argument. Optional arguments for the call include:

- **PROJECT_DIR** - directory of the project; defaults to **CMAKE_SOURCE_DIR**
- **PROJECT_NAME** - name of the project; defaults to **CMAKE_PROJECT_NAME**
- **PROJECT_VER** - version/revision of the project; defaults to "1"
- **SDKCONFIG** - output path of generated *sdkconfig* file; defaults to **PROJECT_DIR/sdkconfig** or **CMAKE_SOURCE_DIR/sdkconfig** depending if **PROJECT_DIR** is set
- **SDKCONFIG_DEFAULTS** - list of files containing default config to use in the build (list must contain full paths); defaults to empty. For each value *filename* in the list, the config from file *filename.target*, if it exists, is also loaded.
- **BUILD_DIR** - directory to place ESP-IDF build-related artifacts, such as generated binaries, text files, components; defaults to **CMAKE_BINARY_DIR**
- **COMPONENTS** - select components to process among the components known by the build system (added via *idf_build_component*). This argument is used to trim the build. Other components are automatically added if they are required in the dependency chain, i.e. the public and private requirements of the components in this list are automatically added, and in turn the public and private requirements of those requirements, so on and so forth. If not specified, all components known to the build system are processed.

```
idf_build_executable(executable)
```

Specify the executable *executable* for ESP-IDF build. This attaches additional targets such as dependencies related to flashing, generating additional binary files, etc. Should be called after `idf_build_process`.

```
idf_build_get_config(var config [GENERATOR_EXPRESSION])
```

Get the value of the specified config. Much like build properties, specifying *GENERATOR_EXPRESSION* will retrieve the generator expression string for that config, instead of the actual value, which can be used with CMake commands that support generator expressions. Actual config values are only known after call to `idf_build_process`, however.

idf-build-properties

These are properties that describe the build. Values of build properties can be retrieved by using the build command `idf_build_get_property`. For example, to get the Python interpreter used for the build:

```
idf_build_get_property(python PYTHON)
message(STATUS "The Python interpreter is: ${python}")
```

- `BUILD_DIR` - build directory; set from `idf_build_process BUILD_DIR` argument
- `BUILD_COMPONENTS` - list of components included in the build; set by `idf_build_process`
- `BUILD_COMPONENT_ALIASES` - list of library alias of components included in the build; set by `idf_build_process`
- `C_COMPILE_OPTIONS` - compile options applied to all components' C source files
- `COMPILE_OPTIONS` - compile options applied to all components' source files, regardless of it being C or C++
- `COMPILE_DEFINITIONS` - compile definitions applied to all component source files
- `CXX_COMPILE_OPTIONS` - compile options applied to all components' C++ source files
- `EXECUTABLE` - project executable; set by call to `idf_build_executable`
- `EXECUTABLE_NAME` - name of project executable without extension; set by call to `idf_build_executable`
- `IDF_PATH` - ESP-IDF path; set from `IDF_PATH` environment variable, if not, inferred from the location of `idf.cmake`
- `IDF_TARGET` - target chip for the build; set from the required target argument for `idf_build_process`
- `IDF_VER` - ESP-IDF version; set from either a version file or the Git revision of the `IDF_PATH` repository
- `INCLUDE_DIRECTORIES` - include directories for all component source files
- `KCONFIGS` - list of Kconfig files found in components in build; set by `idf_build_process`
- `KCONFIG_PROJBUILDS` - list of Kconfig.projbuild files found in components in build; set by `idf_build_process`
- `PROJECT_NAME` - name of the project; set from `idf_build_process PROJECT_NAME` argument
- `PROJECT_DIR` - directory of the project; set from `idf_build_process PROJECT_DIR` argument
- `PROJECT_VER` - version of the project; set from `idf_build_process PROJECT_VER` argument
- `PYTHON` - Python interpreter used for the build; set from `PYTHON` environment variable if available, if not "python" is used
- `SDKCONFIG` - full path to output config file; set from `idf_build_process SDKCONFIG` argument
- `SDKCONFIG_DEFAULTS` - list of files containing default config to use in the build; set from `idf_build_process SDKCONFIG_DEFAULTS` argument
- `SDKCONFIG_HEADER` - full path to C/C++ header file containing component configuration; set by `idf_build_process`
- `SDKCONFIG_CMAKE` - full path to CMake file containing component configuration; set by `idf_build_process`
- `SDKCONFIG_JSON` - full path to JSON file containing component configuration; set by `idf_build_process`
- `SDKCONFIG_JSON_MENUS` - full path to JSON file containing config menus; set by `idf_build_process`

idf-component-commands

```
idf_component_get_property(var component property [GENERATOR_EXPRESSION])
```

Retrieve a specified *component*'s *component property*, *property* and store it in *var* accessible from the current scope. Specifying *GENERATOR_EXPRESSION* will retrieve the generator expression string for that property, instead of the actual value, which can be used with CMake commands that support generator expressions.

```
idf_component_set_property(component property val [APPEND])
```

Set a specified *component*'s *component property*, *property* with value *val*. Specifying *APPEND* will append the specified value to the current value of the property. If the property does not previously exist or it is currently empty, the specified value becomes the first element/member instead.

```
idf_component_register([[SRCS src1 src2 ...] | [[SRC_DIRS dir1 dir2 ...] [EXCLUDE_
↪SRCS src1 src2 ...]])
    [INCLUDE_DIRS dir1 dir2 ...]
    [PRIV_INCLUDE_DIRS dir1 dir2 ...]
    [REQUIRES component1 component2 ...]
    [PRIV_REQUIRES component1 component2 ...]
    [LDFRAGMENTS ldfragment1 ldfragment2 ...]
    [REQUIRED_IDF_TARGETS target1 target2 ...]
    [EMBED_FILES file1 file2 ...]
    [EMBED_TXTFILES file1 file2 ...])
```

Register a component to the build system. Much like the `project()` CMake command, this should be called from the component's `CMakeLists.txt` directly (not through a function or macro) and is recommended to be called before any other command. Here are some guidelines on what commands can **not** be called before `idf_component_register`:

- commands that are not valid in CMake script mode
- custom commands defined in `project_include.cmake`
- build system API commands except `idf_build_get_property`; although consider whether the property may not have been set yet

Commands that set and operate on variables are generally okay to call before `idf_component_register`.

The arguments for `idf_component_register` include:

- **SRCS** - component source files used for creating a static library for the component; if not specified, component is treated as a config-only component and an interface library is created instead.
- **SRC_DIRS**, **EXCLUDE_SRCS** - used to glob source files (`.c`, `.cpp`, `.S`) by specifying directories, instead of specifying source files manually via **SRCS**. Note that this is subject to the *limitations of globbing in CMake*. Source files specified in **EXCLUDE_SRCS** are removed from the globbed files.
- **INCLUDE_DIRS** - paths, relative to the component directory, which will be added to the include search path for all other components which require the current component
- **PRIV_INCLUDE_DIRS** - directory paths, must be relative to the component directory, which will be added to the include search path for this component's source files only
- **REQUIRES** - public component requirements for the component
- **PRIV_REQUIRES** - private component requirements for the component; ignored on config-only components
- **LDFRAGMENTS** - component linker fragment files
- **REQUIRED_IDF_TARGETS** - specify the only target the component supports

The following are used for *embedding data into the component*, and is considered as source files when determining if a component is config-only. This means that even if the component does not specify source files, a static library is still created internally for the component if it specifies either:

- **EMBED_FILES** - binary files to be embedded in the component
- **EMBED_TXTFILES** - text files to be embedded in the component

idf-component-properties

These are properties that describe a component. Values of component properties can be retrieved by using the build command `idf_component_get_property`. For example, to get the directory of the `freertos` component:

```
idf_component_get_property(dir freertos COMPONENT_DIR)
message(STATUS "The 'freertos' component directory is: ${dir}")
```

- `COMPONENT_ALIAS` - alias for `COMPONENT_LIB` used for linking the component to external targets; set by `idf_build_component` and alias library itself is created by `idf_component_register`
- `COMPONENT_DIR` - component directory; set by `idf_build_component`
- `COMPONENT_LIB` - name for created component static/interface library; set by `idf_build_component` and library itself is created by `idf_component_register`
- `COMPONENT_NAME` - name of the component; set by `idf_build_component` based on the component directory name
- `COMPONENT_TYPE` - type of the component, whether `LIBRARY` or `CONFIG_ONLY`. A component is of type `LIBRARY` if it specifies source files or embeds a file
- `EMBED_FILES` - list of files to embed in component; set from `idf_component_register EMBED_FILES` argument
- `EMBED_TXTFILES` - list of text files to embed in component; set from `idf_component_register EMBED_TXTFILES` argument
- `INCLUDE_DIRS` - list of component include directories; set from `idf_component_register INCLUDE_DIRS` argument
- `KCONFIG` - component Kconfig file; set by `idf_build_component`
- `KCONFIG_PROJBUILD` - component Kconfig.projbuild; set by `idf_build_component`
- `LDFRAGMENTS` - list of component linker fragment files; set from `idf_component_register LD-FRAGMENTS` argument
- `PRIV_INCLUDE_DIRS` - list of component private include directories; set from `idf_component_register PRIV_INCLUDE_DIRS` on components of type `LIBRARY`
- `PRIV_REQUIRES` - list of private component dependencies; set from `idf_component_register PRIV_REQUIRES` argument
- `REQUIRED_IDF_TARGETS` - list of targets the component supports; set from `idf_component_register EMBED_TXTFILES` argument
- `REQUIRES` - list of public component dependencies; set from `idf_component_register REQUIRES` argument
- `SRCS` - list of component source files; set from `SRCS` or `SRC_DIRS/EXCLUDE_SRCS` argument of `idf_component_register`

5.4.22 File Globbing & Incremental Builds

The preferred way to include source files in an ESP-IDF component is to list them manually via `SRCS` argument to `idf_component_register`:

```
idf_component_register(SRCS library/a.c library/b.c platform/platform.c
    ...)
```

This preference reflects the [CMake best practice](#) of manually listing source files. This could, however, be inconvenient when there are lots of source files to add to the build. The ESP-IDF build system provides an alternative way for specifying source files using `SRC_DIRS`:

```
idf_component_register(SRC_DIRS library platform
    ...)
```

This uses globbing behind the scenes to find source files in the specified directories. Be aware, however, that if a new source file is added and this method is used, then CMake won't know to automatically re-run and this file won't be added to the build.

The trade-off is acceptable when you're adding the file yourself, because you can trigger a clean build or run `idf.py reconfigure` to manually re-run CMake. However, the problem gets harder when you share your project

with others who may check out a new version using a source control tool like Git...

For components which are part of ESP-IDF, we use a third party Git CMake integration module (/tools/cmake/third_party/GetGitRevisionDescription.cmake) which automatically re-runs CMake any time the repository commit changes. This means if you check out a new ESP-IDF version, CMake will automatically re-run.

For project components (not part of ESP-IDF), there are a few different options:

- If keeping your project file in Git, ESP-IDF will automatically track the Git revision and re-run CMake if the revision changes.
- If some components are kept in a third git repository (not the project repository or ESP-IDF repository), you can add a call to the `git_describe` function in a component CMakeLists file in order to automatically trigger re-runs of CMake when the Git revision changes.
- If not using Git, remember to manually run `idf.py reconfigure` whenever a source file may change.
- To avoid this problem entirely, use `SRCS` argument to `idf_component_register` to list all source files in project components.

The best option will depend on your particular project and its users.

5.4.23 Build System Metadata

For integration into IDEs and other build systems, when CMake runs the build process generates a number of metadata files in the `build/` directory. To regenerate these files, run `cmake` or `idf.py reconfigure` (or any other `idf.py` build command).

- `compile_commands.json` is a standard format JSON file which describes every source file which is compiled in the project. A CMake feature generates this file, and many IDEs know how to parse it.
- `project_description.json` contains some general information about the ESP-IDF project, configured paths, etc.
- `flasher_args.json` contains `esptool.py` arguments to flash the project's binary files. There are also `flash_*_args` files which can be used directly with `esptool.py`. See [Flash arguments](#).
- `CMakeCache.txt` is the CMake cache file which contains other information about the CMake process, toolchain, etc.
- `config/sdkconfig.json` is a JSON-formatted version of the project configuration values.
- `config/kconfig_menus.json` is a JSON-formatted version of the menus shown in `menuconfig`, for use in external IDE UIs.

JSON Configuration Server

A tool called `confserver.py` is provided to allow IDEs to easily integrate with the configuration system logic. `confserver.py` is designed to run in the background and interact with a calling process by reading and writing JSON over process `stdin` & `stdout`.

You can run `confserver.py` from a project via `idf.py confserver` or `ninja confserver`, or a similar target triggered from a different build generator.

The config server outputs human-readable errors and warnings on `stderr` and JSON on `stdout`. On startup, it will output the full values of each configuration item in the system as a JSON dictionary, and the available ranges for values which are range constrained. The same information is contained in `sdkconfig.json`:

```
{ "version": 1, "values": { "ITEM": "value", "ITEM_2": 1024, "ITEM_3": false },  
↪ "ranges" : { "ITEM_2" : [ 0, 32768 ] } }
```

Only visible configuration items are sent. Invisible/disabled items can be parsed from the static `kconfig_menus.json` file which also contains the menu structure and other metadata (descriptions, types, ranges, etc.)

The Configuration Server will then wait for input from the client. The client passes a request to change one or more values, as a JSON object followed by a newline:


```
{ "version": "1", "set": { "SOME_NAME": false, "OTHER_NAME": true } }
```

The Configuration Server will parse this request, update the project `sdkconfig` file, and return a full list of changes:

```
{ "version": 1, "values": { "SOME_NAME": false, "OTHER_NAME": true, "DEPENDS_ON_
↔SOME_NAME": null } }
```

Items which are now invisible/disabled will return value `null`. Any item which is newly visible will return its newly visible current value.

If the range of a config item changes, due to conditional range depending on another value, then this is also sent:

```
{ "version": 1, "values": { "OTHER_NAME": true }, "ranges" : { "HAS_RANGE" : [ 3, 4
↔ ] } }
```

If invalid data is passed, an “error” field is present on the object:

```
{ "version": 1, "values": {}, "error": [ "The following config symbol(s) were not
↔visible so were not updated: NOT_VISIBLE_ITEM" ] }
```

By default, no config changes are written to the `sdkconfig` file. Changes are held in memory until a “save” command is sent:

```
{ "version": 1, "save": null }
```

To reload the config values from a saved file, discarding any changes in memory, a “load” command can be sent:

```
{ "version": 1, "load": null }
```

The value for both “load” and “save” can be a new pathname, or “null” to load/save the previous pathname.

The response to a “load” command is always the full set of config values and ranges, the same as when the server is initially started.

Any combination of “load”, “set”, and “save” can be sent in a single command and commands are executed in that order. Therefore it’s possible to load config from a file, set some config item values and then save to a file in a single command.

Note: The configuration server does not automatically load any changes which are applied externally to the `sd-kconfig` file. Send a “load” command or restart the server if the file is externally edited.

Note: The configuration server does not re-run CMake to regenerate other build files or metadata files after `sd-kconfig` is updated. This will happen automatically the next time CMake or `idf.py` is run.

5.4.24 Build System Internals

Build Scripts

The listfiles for the ESP-IDF build system reside in `/tools/cmake`. The modules which implement core build system functionality are as follows:

- `build.cmake` - Build related commands i.e. build initialization, retrieving/setting build properties, build processing.
- `component.cmake` - Component related commands i.e. adding components, retrieving/setting component properties, registering components.
- `kconfig.cmake` - Generation of configuration files (`sdkconfig`, `sdkconfig.h`, `sdkconfig.cmake`, etc.) from Kconfig files.

- `ldgen.cmake` - Generation of final linker script from linker fragment files.
- `target.cmake` - Setting build target and toolchain file.
- `utilities.cmake` - Miscellaneous helper commands.

Aside from these files, there are two other important CMake scripts in `/tools/cmake`:

- `idf.cmake` - Sets up the build and includes the core modules listed above. Included in CMake projects in order to access ESP-IDF build system functionality.
- `project.cmake` - Includes `idf.cmake` and provides a custom `project()` command that takes care of all the heavy lifting of building an executable. Included in the top-level `CMakeLists.txt` of standard ESP-IDF projects.

The rest of the files in `/tools/cmake` are support or third-party scripts used in the build process.

Build Process

This section describes the standard ESP-IDF application build process. The build process can be broken down roughly into four phases:



Fig. 3: ESP-IDF Build System Process

Initialization

This phase sets up necessary parameters for the build.

- **Upon inclusion of `idf.cmake` in `project.cmake`, the following steps are performed:**
 - Set `IDF_PATH` from environment variable or inferred from path to `project.cmake` included in the top-level `CMakeLists.txt`.
 - Add `/tools/cmake` to `CMAKE_MODULE_PATH` and include core modules plus the various helper/third-party scripts.
 - Set build tools/executables such as default Python interpreter.
 - Get ESP-IDF git revision and store as `IDF_VER`.
 - Set global build specifications i.e. compile options, compile definitions, include directories for all components in the build.
 - Add components in `components` to the build.
- **The initial part of the custom `project()` command performs the following steps:**
 - Set `IDF_TARGET` from environment variable or CMake cache and the corresponding `CMAKE_TOOLCHAIN_FILE` to be used.
 - Add components in `EXTRA_COMPONENTS_DIRS` to the build.
 - Prepare arguments for calling command `idf_build_process()` from variables such as `COMPONENTS/EXCLUDE_COMPONENTS`, `SDKCONFIG`, `SDKCONFIG_DEFAULTS`.

The call to `idf_build_process()` command marks the end of this phase.

Enumeration

This phase builds a final list of components to be processed in the build, and is performed in the first half of `idf_build_process()`.

- Retrieve each component's public and private requirements. A child process is created which executes each component's CMakeLists.txt in script mode. The values of `idf_component_register` `REQUIRES` and `PRIV_REQUIRES` argument is returned to the parent build process. This is called early expansion. The variable `CMAKE_BUILD_EARLY_EXPANSION` is defined during this step.
- Recursively include components based on public and private requirements.

Processing

This phase processes the components in the build, and is the second half of `idf_build_process()`.

- Load project configuration from `sdkconfig` file and generate an `sdkconfig.cmake` and `sdkconfig.h` header. These define configuration variables/macros that are accessible from the build scripts and C/C++ source/header files, respectively.
- Include each component's `project_include.cmake`.
- Add each component as a subdirectory, processing its CMakeLists.txt. The component CMakeLists.txt calls the registration command, `idf_component_register` which adds source files, include directories, creates component library, links dependencies, etc.

Finalization

This phase is everything after `idf_build_process()`.

- Create executable and link the component libraries to it.
- Generate project metadata files such as `project_description.json` and display relevant information about the project built.

Browse </tools/cmake/project.cmake> for more details.

5.4.25 Migrating from ESP-IDF GNU Make System

Some aspects of the CMake-based ESP-IDF build system are very similar to the older GNU Make-based system. The developer needs to provide values the include directories, source files etc. There is a syntactical difference, however, as the developer needs to pass these as arguments to the registration command, `idf_component_register`.

Automatic Conversion Tool

An automatic project conversion tool is available in /tools/cmake/convert_to_cmake.py. Run this command line tool with the path to a project like this:

```
$IDF_PATH/tools/cmake/convert_to_cmake.py /path/to/project_dir
```

The project directory must contain a Makefile, and GNU Make (`make`) must be installed and available on the `PATH`.

The tool will convert the project Makefile and any component `component.mk` files to their equivalent CMakeLists.txt files.

It does so by running `make` to expand the ESP-IDF build system variables which are set by the build, and then producing equivalent CMakeLists files to set the same variables.

Important: When the conversion tool converts a `component.mk` file, it doesn't determine what other components that component depends on. This information needs to be added manually by editing the new component CMakeLists.txt file and adding `REQUIRES` and/or `PRIV_REQUIRES` clauses. Otherwise, source files in the component will fail to compile as headers from other components are not found. See [Component Requirements](#).

The conversion tool is not capable of dealing with complex Makefile logic or unusual targets. These will need to be converted by hand.

No Longer Available in CMake

Some features are significantly different or removed in the CMake-based system. The following variables no longer exist in the CMake-based build system:

- `COMPONENT_BUILD_DIR`: Use `CMAKE_CURRENT_BINARY_DIR` instead.
- `COMPONENT_LIBRARY`: Defaulted to `$(COMPONENT_NAME).a`, but the library name could be overridden by the component. The name of the component library can no longer be overridden by the component.
- `CC`, `LD`, `AR`, `OBJCOPY`: Full paths to each tool from the gcc xtensa cross-toolchain. Use `CMAKE_C_COMPILER`, `CMAKE_C_LINK_EXECUTABLE`, `CMAKE_OBJCOPY`, etc instead. [Full list here](#).
- `HOSTCC`, `HOSTLD`, `HOSTAR`: Full names of each tool from the host native toolchain. These are no longer provided, external projects should detect any required host toolchain manually.
- `COMPONENT_ADD_LDFLAGS`: Used to override linker flags. Use the CMake [target_link_libraries](#) command instead.
- `COMPONENT_ADD_LINKER_DEPS`: List of files that linking should depend on. [target_link_libraries](#) will usually infer these dependencies automatically. For linker scripts, use the provided custom CMake function `target_linker_scripts`.
- `COMPONENT_SUBMODULES`: No longer used, the build system will automatically enumerate all submodules in the ESP-IDF repository.
- `COMPONENT_EXTRA_INCLUDES`: Used to be an alternative to `COMPONENT_PRIV_INCLUDEDIRS` for absolute paths. Use `COMPONENT_PRIV_INCLUDEDIRS` argument to `idf_component_register` for all cases now (can be relative or absolute).
- `COMPONENT_OBJS`: Previously, component sources could be specified as a list of object files. Now they can be specified as a list of source files via `SRC` argument to `idf_component_register`.
- `COMPONENT_OBJEXCLUDE`: Has been replaced with `EXCLUDE_SRCS` argument to `idf_component_register`. Specify source files (as absolute paths or relative to component directory), instead.
- `COMPONENT_EXTRA_CLEAN`: Set property `ADDITIONAL_MAKE_CLEAN_FILES` instead but note *CMake has some restrictions around this functionality*.
- `COMPONENT_OWNBUILDTARGET` & `COMPONENT_OWNCLEANTARGET`: Use CMake [ExternalProject](#) instead. See [Fully Overriding The Component Build Process](#) for full details.
- `COMPONENT_CONFIG_ONLY`: Call `idf_component_register` without any arguments instead. See [Configuration-Only Components](#).
- `CFLAGS`, `CPPFLAGS`, `CXXFLAGS`: Use equivalent CMake commands instead. See [Controlling Component Compilation](#).

No Default Values

Unlike in the legacy Make-based build system, the following have no default values:

- Source directories (`COMPONENT_SRCDIRS` variable in Make, `SRC_DIRS` argument to `idf_component_register` in CMake)
- Include directories (`COMPONENT_ADD_INCLUDEDIRS` variable in Make, `INCLUDE_DIRS` argument to `idf_component_register` in CMake)

No Longer Necessary

- In the legacy Make-based build system, it is required to also set `COMPONENT_SRCDIRS` if `COMPONENT_SRCS` is set. In CMake, the equivalent is not necessary i.e. specifying `SRC_DIRS` to `idf_component_register` if `SRC` is also specified (in fact, `SRC` is ignored if `SRC_DIRS` is specified).

Flashing from make

`make flash` and similar targets still work to build and flash. However, project `sdkconfig` no longer specifies serial port and baud rate. Environment variables can be used to override these. See [Flashing with ninja or make](#) for more details.

5.5 Build System (Legacy GNU Make)

Note: Since ESP-IDF V4.0, the default build system is based on CMake. This documentation is for the legacy build system based on GNU Make. Support for this build system may be removed in future major releases.

This document explains the legacy GNU Make Espressif IoT Development Framework build system and the concept of “components”

Read this document if you want to know how to organise an ESP-IDF project using GNU Make build system.

We recommend using the [esp-idf-template](#) project as a starting point for your project.

5.5.1 Using the Build System

The esp-idf README file contains a description of how to use the build system to build your project.

5.5.2 Overview

An ESP-IDF project can be seen as an amalgamation of a number of components. For example, for a webserver that shows the current humidity, there could be:

- The ESP32 base libraries (libc, rom bindings etc)
- The Wi-Fi drivers
- A TCP/IP stack
- The FreeRTOS operating system
- A webserver
- A driver for the humidity sensor
- Main code tying it all together

ESP-IDF makes these components explicit and configurable. To do that, when a project is compiled, the build environment will look up all the components in the ESP-IDF directories, the project directories and (optionally) in additional custom component directories. It then allows the user to configure the ESP-IDF project using a text-based menu system to customize each component. After the components in the project are configured, the build process will compile the project.

Concepts

- A “project” is a directory that contains all the files and configuration to build a single “app” (executable), as well as additional supporting output such as a partition table, data/filesystem partitions, and a bootloader.
- “Project configuration” is held in a single file called `sdkconfig` in the root directory of the project. This configuration file is modified via `make menuconfig` to customise the configuration of the project. A single project contains exactly one project configuration.
- An “app” is an executable which is built by esp-idf. A single project will usually build two apps - a “project app” (the main executable, ie your custom firmware) and a “bootloader app” (the initial bootloader program which launches the project app).
- “components” are modular pieces of standalone code which are compiled into static libraries (.a files) and linked into an app. Some are provided by esp-idf itself, others may be sourced from other places.

Some things are not part of the project:

- “ESP-IDF” is not part of the project. Instead it is standalone, and linked to the project via the `IDF_PATH` environment variable which holds the path of the `esp-idf` directory. This allows the IDF framework to be decoupled from your project.
- The toolchain for compilation is not part of the project. The toolchain should be installed in the system command line `PATH`, or the path to the toolchain can be set as part of the compiler prefix in the project configuration.

Example Project

An example project directory tree might look like this:

```

- myProject/
  - Makefile
  - sdkconfig
  - components/
    - component1/
      - component.mk
      - Kconfig
      - src1.c
    - component2/
      - component.mk
      - Kconfig
      - src1.c
      - include/
        - component2.h
  - main/
    - src1.c
    - src2.c
    - component.mk
  - build/

```

This example “myProject” contains the following elements:

- A top-level project Makefile. This Makefile sets the `PROJECT_NAME` variable and (optionally) defines other project-wide make variables. It includes the core `$(IDF_PATH)/make/project.mk` makefile which implements the rest of the ESP-IDF build system.
- “sdkconfig” project configuration file. This file is created/updated when “make menuconfig” runs, and holds configuration for all of the components in the project (including esp-idf itself). The “sdkconfig” file may or may not be added to the source control system of the project.
- Optional “components” directory contains components that are part of the project. A project does not have to contain custom components of this kind, but it can be useful for structuring reusable code or including third party components that aren’t part of ESP-IDF.
- “main” directory is a special “pseudo-component” that contains source code for the project itself. “main” is a default name, the Makefile variable `COMPONENT_DIRS` includes this component but you can modify this variable (or set `EXTRA_COMPONENT_DIRS`) to look for components in other places.
- “build” directory is where build output is created. After the make process is run, this directory will contain interim object files and libraries as well as final binary output files. This directory is usually not added to source control or distributed with the project source code.

Component directories contain a component makefile - `component.mk`. This may contain variable definitions to control the build process of the component, and its integration into the overall project. See [Component Makefiles](#) for more details.

Each component may also include a `Kconfig` file defining the *component configuration* options that can be set via the project configuration. Some components may also include `Kconfig.projbuild` and `Makefile.projbuild` files, which are special files for *overriding parts of the project*.

Project Makefiles

Each project has a single Makefile that contains build settings for the entire project. By default, the project Makefile can be quite minimal.

Minimal Example Makefile

```

PROJECT_NAME := myProject

include $(IDF_PATH)/make/project.mk

```

Mandatory Project Variables

- `PROJECT_NAME`: Name of the project. Binary output files will use this name - ie `myProject.bin`, `myProject.elf`.

Optional Project Variables These variables all have default values that can be overridden for custom behaviour. Look in `make/project.mk` for all of the implementation details.

- `PROJECT_PATH`: Top-level project directory. Defaults to the directory containing the Makefile. Many other project variables are based on this variable. The project path cannot contain spaces.
- `BUILD_DIR_BASE`: The build directory for all objects/libraries/binaries. Defaults to `$(PROJECT_PATH)/build`.
- `COMPONENT_DIRS`: Directories to search for components. Defaults to `$(IDF_PATH)/components`, `$(PROJECT_PATH)/components`, `$(PROJECT_PATH)/main` and `EXTRA_COMPONENT_DIRS`. Override this variable if you don't want to search for components in these places.
- `EXTRA_COMPONENT_DIRS`: Optional list of additional directories to search for components.
- `COMPONENTS`: A list of component names to build into the project. Defaults to all components found in the `COMPONENT_DIRS` directories.
- `EXCLUDE_COMPONENTS`: Optional list of component names to exclude during the build process. Note that this decreases build time, but not binary size.
- `TEST_EXCLUDE_COMPONENTS`: Optional list of component names to exclude during the build process of unit tests.

Any paths in these Makefile variables should be absolute paths. You can convert relative paths using `$(PROJECT_PATH)/xxx`, `$(IDF_PATH)/xxx`, or use the Make function `$(abspath xxx)`.

These variables should all be set before the line `include $(IDF_PATH)/make/project.mk` in the Makefile.

Component Makefiles

Each project contains one or more components, which can either be part of `esp-idf` or added from other component directories.

A component is any directory that contains a `component.mk` file.

Searching for Components

The list of directories in `COMPONENT_DIRS` is searched for the project's components. Directories in this list can either be components themselves (ie they contain a `component.mk` file), or they can be top-level directories whose subdirectories are components.

Running the `make list-components` target dumps many of these variables and can help debug the discovery of component directories.

Multiple components with the same name When `esp-idf` is collecting all the components to compile, it will do this in the order specified by `COMPONENT_DIRS`; by default, this means the `idf` components first, the project components second and optionally the components in `EXTRA_COMPONENT_DIRS` last. If two or more of these directories contain component subdirectories with the same name, the component in the last place searched is used. This allows, for example, overriding `esp-idf` components with a modified version by simply copying the component from the `esp-idf` component directory to the project component tree and then modifying it there. If used in this way, the `esp-idf` directory itself can remain untouched.

Minimal Component Makefile The minimal `component.mk` file is an empty file(!). If the file is empty, the default component behaviour is set:

- All source files in the same directory as the makefile (`*.c`, `*.cpp`, `*.cc`, `*.S`) will be compiled into the component library
- A sub-directory "include" will be added to the global include search path for all other components.
- The component library will be linked into the project app.

See *example component makefiles* for more complete component makefile examples.

Note that there is a difference between an empty `component.mk` file (which invokes default component build behaviour) and no `component.mk` file (which means no default component build behaviour will occur.) It is possible for a component to have no `component.mk` file, if it only contains other files which influence the project configuration or build process.

Preset Component Variables The following component-specific variables are available for use inside `component.mk`, but should not be modified:

- `COMPONENT_PATH`: The component directory. Evaluates to the absolute path of the directory containing `component.mk`. The component path cannot contain spaces.
- `COMPONENT_NAME`: Name of the component. Defaults to the name of the component directory.
- `COMPONENT_BUILD_DIR`: The component build directory. Evaluates to the absolute path of a directory inside `$(BUILD_DIR_BASE)` where this component's source files are to be built. This is also the Current Working Directory any time the component is being built, so relative paths in make targets, etc. will be relative to this directory.
- `COMPONENT_LIBRARY`: Name of the static library file (relative to the component build directory) that will be built for this component. Defaults to `$(COMPONENT_NAME).a`.

The following variables are set at the project level, but exported for use in the component build:

- `PROJECT_NAME`: Name of the project, as set in project Makefile
- `PROJECT_PATH`: Absolute path of the project directory containing the project Makefile.
- `COMPONENTS`: Name of all components that are included in this build.
- `CONFIG_*`: Each value in the project configuration has a corresponding variable available in make. All names begin with `CONFIG_`.
- `CC, LD, AR, OBJCOPY`: Full paths to each tool from the gcc xtensa cross-toolchain.
- `HOSTCC, HOSTLD, HOSTAR`: Full names of each tool from the host native toolchain.
- `IDF_VER`: ESP-IDF version, retrieved from either `$(IDF_PATH)/version.txt` file (if present) else using git command `git describe`. Recommended format here is single liner that specifies major IDF release version, e.g. `v2.0` for a tagged release or `v2.0-275-g0efaa4f` for an arbitrary commit. Application can make use of this by calling `esp_get_idf_version()`.
- `IDF_VERSION_MAJOR, IDF_VERSION_MINOR, IDF_VERSION_PATCH`: Components of ESP-IDF version, to be used in conditional expressions. Note that this information is less precise than that provided by `IDF_VER` variable. `v4.0-dev-*`, `v4.0-beta1`, `v4.0-rc1` and `v4.0` will all have the same values of `ESP_IDF_VERSION_*` variables, but different `IDF_VER` values.
- `PROJECT_VER`: Project version.
 - If `PROJECT_VER` variable is set in project Makefile file, its value will be used.
 - Else, if the `$(PROJECT_PATH)/version.txt` exists, its contents will be used as `PROJECT_VER`.
 - Else, if the project is located inside a Git repository, the output of `git describe` will be used.
 - Otherwise, `PROJECT_VER` will be "1".

If you modify any of these variables inside `component.mk` then this will not prevent other components from building but it may make your component hard to build and/or debug.

Optional Project-Wide Component Variables The following variables can be set inside `component.mk` to control build settings across the entire project:

- `COMPONENT_ADD_INCLUDEDIRS`: Paths, relative to the component directory, which will be added to the include search path for all components in the project. Defaults to `include` if not overridden. If an include directory is only needed to compile this specific component, add it to `COMPONENT_PRIV_INCLUDEDIRS` instead.
- `COMPONENT_ADD_LDFLAGS`: Add linker arguments to the `LDFLAGS` for the app executable. Defaults to `-l$(COMPONENT_NAME)`. If adding pre-compiled libraries to this directory, add them as absolute paths - ie `$(COMPONENT_PATH)/libwhatever.a`
- `COMPONENT_DEPENDS`: Optional list of component names that should be compiled before this component. This is not necessary for link-time dependencies, because all component include directories are available at all times. It is necessary if one component generates an include file which you then want to include in another component. Most components do not need to set this variable.

- `COMPONENT_ADD_LINKER_DEPS`: Optional list of component-relative paths to files which should trigger a re-link of the ELF file if they change. Typically used for linker script files and binary libraries. Most components do not need to set this variable.

The following variable only works for components that are part of esp-idf itself:

- `COMPONENT_SUBMODULES`: Optional list of git submodule paths (relative to `COMPONENT_PATH`) used by the component. These will be checked (and initialised if necessary) by the build process. This variable is ignored if the component is outside the `IDF_PATH` directory.

Optional Component-Specific Variables The following variables can be set inside `component.mk` to control the build of that component:

- `COMPONENT_PRIV_INCLUDEDIRS`: Directory paths, must be relative to the component directory, which will be added to the include search path for this component's source files only.
- `COMPONENT_EXTRA_INCLUDES`: Any extra include paths used when compiling the component's source files. These will be prefixed with `'-I'` and passed as-is to the compiler. Similar to the `COMPONENT_PRIV_INCLUDEDIRS` variable, except these paths are not expanded relative to the component directory.
- `COMPONENT_SRCDIRS`: Directory paths, must be relative to the component directory, which will be searched for source files (`*.cpp`, `*.c`, `*.S`). Defaults to `'.'`, ie the component directory itself. Override this to specify a different list of directories which contain source files.
- `COMPONENT_OBJS`: Object files to compile. Default value is a `.o` file for each source file that is found in `COMPONENT_SRCDIRS`. Overriding this list allows you to exclude source files in `COMPONENT_SRCDIRS` that would otherwise be compiled. See *Specifying source files*
- `COMPONENT_EXTRA_CLEAN`: Paths, relative to the component build directory, of any files that are generated using custom make rules in the `component.mk` file and which need to be removed as part of `make clean`. See *Source Code Generation* for an example.
- `COMPONENT_OWNBUILDTARGET` & `COMPONENT_OWNCLEANTARGET`: These targets allow you to fully override the default build behaviour for the component. See *Fully Overriding The Component Makefile* for more details.
- `COMPONENT_CONFIG_ONLY`: If set, this flag indicates that the component produces no built output at all (ie `COMPONENT_LIBRARY` is not built), and most other component variables are ignored. This flag is used for IDF internal components which contain only `Kconfig.projbuild` and/or `Makefile.projbuild` files to configure the project, but no source files.
- `CFLAGS`: Flags passed to the C compiler. A default set of `CFLAGS` is defined based on project settings. Component-specific additions can be made via `CFLAGS +=`. It is also possible (although not recommended) to override this variable completely for a component.
- `CPPFLAGS`: Flags passed to the C preprocessor (used for `.c`, `.cpp` and `.S` files). A default set of `CPPFLAGS` is defined based on project settings. Component-specific additions can be made via `CPPFLAGS +=`. It is also possible (although not recommended) to override this variable completely for a component.
- `CXXFLAGS`: Flags passed to the C++ compiler. A default set of `CXXFLAGS` is defined based on project settings. Component-specific additions can be made via `CXXFLAGS +=`. It is also possible (although not recommended) to override this variable completely for a component.
- `COMPONENT_ADD_LDFRAGMENTS`: Paths to linker fragment files for the linker script generation functionality. See *Linker Script Generation*.

To apply compilation flags to a single source file, you can add a variable override as a target, ie:

```
apps/dhcpserver.o: CFLAGS += -Wno-unused-variable
```

This can be useful if there is upstream code that emits warnings.

Component Configuration

Each component can also have a `Kconfig` file, alongside `component.mk`. This contains configuration settings to add to the “make menuconfig” for this component.

These settings are found under the “Component Settings” menu when `menuconfig` is run.

To create a component KConfig file, it is easiest to start with one of the KConfig files distributed with esp-idf.

For an example, see [Adding conditional configuration](#).

Preprocessor Definitions

ESP-IDF build systems adds the following C preprocessor definitions on the command line:

- `ESP_PLATFORM` —Can be used to detect that build happens within ESP-IDF.
- `IDF_VER` —ESP-IDF version, see [Preset Component Variables](#) for more details.

Build Process Internals

Top Level: Project Makefile

- “make” is always run from the project directory and the project makefile, typically named `Makefile`.
- The project makefile sets `PROJECT_NAME` and optionally customises other *optional project variables*
- The project makefile includes `$(IDF_PATH)/make/project.mk` which contains the project-level Make logic.
- `project.mk` fills in default project-level make variables and includes make variables from the project configuration. If the generated makefile containing project configuration is out of date, then it is regenerated (via targets in `project_config.mk`) and then the make process restarts from the top.
- `project.mk` builds a list of components to build, based on the default component directories or a custom list of components set in *optional project variables*.
- Each component can set some *optional project-wide component variables*. These are included via generated makefiles named `component_project_vars.mk` - there is one per component. These generated makefiles are included into `project.mk`. If any are missing or out of date, they are regenerated (via a recursive make call to the component makefile) and then the make process restarts from the top.
- `Makefile.projbuild` files from components are included into the make process, to add extra targets or configuration.
- By default, the project makefile also generates top-level build & clean targets for each component and sets up `app` and `clean` targets to invoke all of these sub-targets.
- In order to compile each component, a recursive make is performed for the component makefile.

To better understand the project make process, have a read through the `project.mk` file itself.

Second Level: Component Makefiles

- Each call to a component makefile goes via the `$(IDF_PATH)/make/component_wrapper.mk` wrapper makefile.
- This component wrapper includes all `component Makefile.componentbuild` files, making any recipes, variables etc in these files available to every component.
- The `component_wrapper.mk` is called with the current directory set to the component build directory, and the `COMPONENT_MAKEFILE` variable is set to the absolute path to `component.mk`.
- `component_wrapper.mk` sets default values for all *component variables*, then includes the `component.mk` file which can override or modify these.
- If `COMPONENT_OWNBUILDTARGET` and `COMPONENT_OWNCLEANTARGET` are not defined, default build and clean targets are created for the component’s source files and the prerequisite `COMPONENT_LIBRARY` static library file.
- The `component_project_vars.mk` file has its own target in `component_wrapper.mk`, which is evaluated from `project.mk` if this file needs to be rebuilt due to changes in the component makefile or the project configuration.

To better understand the component make process, have a read through the `component_wrapper.mk` file and some of the `component.mk` files included with esp-idf.

Running Make Non-Interactively

When running `make` in a situation where you don't want interactive prompts (for example: inside an IDE or an automated build system) append `BATCH_BUILD=1` to the make arguments (or set it as an environment variable).

Setting `BATCH_BUILD` implies the following:

- Verbose output (same as `V=1`, see below). If you don't want verbose output, also set `V=0`.
- If the project configuration is missing new configuration items (from new components or esp-idf updates) then the project use the default values, instead of prompting the user for each item.
- If the build system needs to invoke `menuconfig`, an error is printed and the build fails.

Advanced Make Targets

- `make app`, `make bootloader`, `make partition table` can be used to build only the app, bootloader, or partition table from the project as applicable.
- `make erase_flash` and `make erase_otadata` will use `esptool.py` to erase the entire flash chip and the OTA selection setting from the flash chip, respectively.
- `make size` prints some size information about the app. `make size-components` and `make size-files` are similar targets which print more detailed per-component or per-source-file information, respectively.

Debugging The Make Process

Some tips for debugging the esp-idf build system:

- Appending `V=1` to the make arguments (or setting it as an environment variable) will cause make to echo all commands executed, and also each directory as it is entered for a sub-make.
- Running `make -w` will cause make to echo each directory as it is entered for a sub-make - same as `V=1` but without also echoing all commands.
- Running `make --trace` (possibly in addition to one of the above arguments) will print out every target as it is built, and the dependency which caused it to be built.
- Running `make -p` prints a (very verbose) summary of every generated target in each makefile.

For more debugging tips and general make information, see the *GNU Make Manual*.

Warning On Undefined Variables By default, the build process will print a warning if an undefined variable is referenced (like `$(DOES_NOT_EXIST)`). This can be useful to find errors in variable names.

If you don't want this behaviour, it can be disabled in `menuconfig`'s top level menu under *SDK tool configuration*.

Note that this option doesn't trigger a warning if `ifdef` or `ifndef` are used in Makefiles.

Overriding Parts of the Project

Makefile.projbuild For components that have build requirements that must be evaluated in the top-level project make pass, you can create a file called `Makefile.projbuild` in the component directory. This makefile is included when `project.mk` is evaluated.

For example, if your component needs to add to `CFLAGS` for the entire project (not just for its own source files) then you can set `CFLAGS +=` in `Makefile.projbuild`.

`Makefile.projbuild` files are used heavily inside esp-idf, for defining project-wide build features such as `esptool.py` command line arguments and the bootloader "special app".

Note that `Makefile.projbuild` isn't necessary for the most common component uses - such as adding include directories to the project, or `LD_FLAGS` to the final linking step. These values can be customised via the component `.mk` file itself. See *Optional Project-Wide Component Variables* for details.

Take care when setting variables or targets in this file. As the values are included into the top-level project makefile pass, they can influence or break functionality across all components!

KConfig.projbuild This is an equivalent to `Makefile.projbuild` for *component configuration* KConfig files. If you want to include configuration options at the top-level of `menuconfig`, rather than inside the “Component Configuration” sub-menu, then these can be defined in the `KConfig.projbuild` file alongside the `component.mk` file.

Take care when adding configuration values in this file, as they will be included across the entire project configuration. Where possible, it’s generally better to create a KConfig file for *component configuration*.

Makefile.componentbuild For components that e.g. include tools to generate source files from other files, it is necessary to be able to add recipes, macros or variable definitions into the component build process of every components. This is done by having a `Makefile.componentbuild` in a component directory. This file gets included in `component_wrapper.mk`, before the `component.mk` of the component is included. As with the `Makefile.projbuild`, take care with these files: as they’re included in each component build, a `Makefile.componentbuild` error may only show up when compiling an entirely different component.

Configuration-Only Components Some special components which contain no source files, only `Kconfig.projbuild` and `Makefile.projbuild`, may set the flag `COMPONENT_CONFIG_ONLY` in the `component.mk` file. If this flag is set, most other component variables are ignored and no build step is run for the component.

Example Component Makefiles

Because the build environment tries to set reasonable defaults that will work most of the time, `component.mk` can be very small or even empty (see [Minimal Component Makefile](#)). However, overriding *component variables* is usually required for some functionality.

Here are some more advanced examples of `component.mk` makefiles:

Adding source directories By default, sub-directories are ignored. If your project has sources in sub-directories instead of in the root of the component then you can tell that to the build system by setting `COMPONENT_SRCDIRS`:

```
COMPONENT_SRCDIRS := src1 src2
```

This will compile all source files in the `src1/` and `src2/` sub-directories instead.

Specifying source files The standard `component.mk` logic adds all `.S` and `.c` files in the source directories as sources to be compiled unconditionally. It is possible to circumvent that logic and hard-code the objects to be compiled by manually setting the `COMPONENT_OBJS` variable to the name of the objects that need to be generated:

```
COMPONENT_OBJS := file1.o file2.o thing/filea.o thing/fileb.o anotherthing/main.o
COMPONENT_SRCDIRS := . thing anotherthing
```

Note that `COMPONENT_SRCDIRS` must be set as well.

Adding conditional configuration The configuration system can be used to conditionally compile some files depending on the options selected in `make menuconfig`. For this, ESP-IDF has the `compile_only_if` and `compile_only_if_not` macros:

Kconfig:

```
config FOO_ENABLE_BAR
    bool "Enable the BAR feature."
    help
        This enables the BAR feature of the FOO component.
```

component.mk:

```
$(call compile_only_if,$(CONFIG_FOO_ENABLE_BAR),bar.o)
```

As can be seen in the example, the `compile_only_if` macro takes a condition and a list of object files as parameters. If the condition is true (in this case: if the BAR feature is enabled in menuconfig) the object files (in this case: bar.o) will always be compiled. The opposite goes as well: if the condition is not true, bar.o will never be compiled. `compile_only_if_not` does the opposite: compile if the condition is false, not compile if the condition is true.

This can also be used to select or stub out an implementation, as such:

Kconfig:

```
config ENABLE_LCD_OUTPUT
    bool "Enable LCD output."
    help
        Select this if your board has a LCD.

config ENABLE_LCD_CONSOLE
    bool "Output console text to LCD"
    depends on ENABLE_LCD_OUTPUT
    help
        Select this to output debugging output to the lcd

config ENABLE_LCD_PLOT
    bool "Output temperature plots to LCD"
    depends on ENABLE_LCD_OUTPUT
    help
        Select this to output temperature plots
```

component.mk:

```
# If LCD is enabled, compile interface to it, otherwise compile dummy interface
$(call compile_only_if,$(CONFIG_ENABLE_LCD_OUTPUT),lcd-real.o lcd-spi.o)
$(call compile_only_if_not,$(CONFIG_ENABLE_LCD_OUTPUT),lcd-dummy.o)

#We need font if either console or plot is enabled
$(call compile_only_if,$(or $(CONFIG_ENABLE_LCD_CONSOLE),$(CONFIG_ENABLE_LCD_
↪PLOT)), font.o)
```

Note the use of the Make ‘or’ function to include the font file. Other substitution functions, like ‘and’ and ‘if’ will also work here. Variables that do not come from menuconfig can also be used: ESP-IDF uses the default Make policy of judging a variable which is empty or contains only whitespace to be false while a variable with any non-whitespace in it is true.

(Note: Older versions of this document advised conditionally adding object file names to `COMPONENT_OBJS`. While this still is possible, this will only work when all object files for a component are named explicitly, and will not clean up deselected object files in a `make clean` pass.)

Source Code Generation Some components will have a situation where a source file isn’t supplied with the component itself but has to be generated from another file. Say our component has a header file that consists of the converted binary data of a BMP file, converted using a hypothetical tool called `bmp2h`. The header file is then included in as C source file called `graphics_lib.c`:

```
COMPONENT_EXTRA_CLEAN := logo.h

graphics_lib.o: logo.h

logo.h: $(COMPONENT_PATH)/logo.bmp
    bmp2h -i $^ -o $@
```

In this example, `graphics_lib.o` and `logo.h` will be generated in the current directory (the build directory) while `logo.bmp` comes with the component and resides under the component path. Because `logo.h` is a generated file, it needs to be cleaned when `make clean` is called which why it is added to the `COMPONENT_EXTRA_CLEAN` variable.

Cosmetic Improvements Because `logo.h` is a generated file, it needs to be cleaned when `make clean` is called which why it is added to the `COMPONENT_EXTRA_CLEAN` variable.

Adding `logo.h` to the `graphics_lib.o` dependencies causes it to be generated before `graphics_lib.c` is compiled.

If a source file in another component included `logo.h`, then this component's name would have to be added to the other component's `COMPONENT_DEPENDS` list to ensure that the components were built in-order.

Embedding Binary Data Sometimes you have a file with some binary or text data that you'd like to make available to your component - but you don't want to reformat the file as C source.

You can set a variable `COMPONENT_EMBED_FILES` in `component.mk`, giving the names of the files to embed in this way:

```
COMPONENT_EMBED_FILES := server_root_cert.der
```

Or if the file is a string, you can use the variable `COMPONENT_EMBED_TXTFILES`. This will embed the contents of the text file as a null-terminated string:

```
COMPONENT_EMBED_TXTFILES := server_root_cert.pem
```

The file's contents will be added to the `.rodata` section in flash, and are available via symbol names as follows:

```
extern const uint8_t server_root_cert_pem_start[] asm("_binary_server_root_cert_
↪pem_start");
extern const uint8_t server_root_cert_pem_end[]   asm("_binary_server_root_cert_
↪pem_end");
```

The names are generated from the full name of the file, as given in `COMPONENT_EMBED_FILES`. Characters `/`, `.`, etc. are replaced with underscores. The `_binary` prefix in the symbol name is added by `objcopy` and is the same for both text and binary files.

For an example of using this technique, see [protocols/https_request](#) - the certificate file contents are loaded from the text `.pem` file at compile time.

Code and Data Placements

ESP-IDF has a feature called linker script generation that enables components to define where its code and data will be placed in memory through linker fragment files. These files are processed by the build system, and is used to augment the linker script used for linking app binary. See [Linker Script Generation](#) for a quick start guide as well as a detailed discussion of the mechanism.

Fully Overriding The Component Makefile

Obviously, there are cases where all these recipes are insufficient for a certain component, for example when the component is basically a wrapper around another third-party component not originally intended to be compiled under this build system. In that case, it's possible to forego the `esp-idf` build system entirely by setting `COMPONENT_OWNBUILDTARGET` and possibly `COMPONENT_OWNCLEANTARGET` and defining your own targets named `build` and `clean` in `component.mk` target. The build target can do anything as long as it creates `$(COMPONENT_LIBRARY)` for the project make process to link into the app binary.

(Actually, even this is not strictly necessary - if the `COMPONENT_ADD_LDFLAGS` variable is overridden then the component can instruct the linker to link other binaries instead.)

Note: When using an external build process with PSRAM, remember to add `-mfix-esp32-psram-cache-issue` to the C compiler arguments. See [CONFIG_SPIRAM_CACHE_WORKAROUND](#) for details of this flag.

Custom sdkconfig defaults

For example projects or other projects where you don't want to specify a full `sdkconfig` configuration, but you do want to override some key values from the `esp-idf` defaults, it is possible to create a file `sdkconfig.defaults` in the project directory. This file will be used when running `make defconfig`, or creating a new config from scratch.

To override the name of this file, set the `SDKCONFIG_DEFAULTS` environment variable.

Save flash arguments

There're some scenarios that we want to flash the target board without IDF. For this case we want to save the built binaries, `esptool.py` and `esptool write_flash` arguments. It's simple to write a script to save binaries and `esptool.py`. We can use command `make print_flash_cmd`, it will print the flash arguments:

```
--flash_mode dio --flash_freq 40m --flash_size detect 0x1000 bootloader/bootloader.  
↪bin 0x10000 example_app.bin 0x8000 partition_table_unit_test_app.bin
```

Then use flash arguments as the arguemnts for `esptool write_flash` arguments:

```
python esptool.py --chip esp32 --port /dev/ttyUSB0 --baud 921600 --before default_  
↪reset --after hard_reset write_flash -z --flash_mode dio --flash_freq 40m --  
↪flash_size detect 0x1000 bootloader/bootloader.bin 0x10000 example_app.bin  
↪0x8000 partition_table_unit_test_app.bin
```

5.5.3 Building the Bootloader

The bootloader is built by default as part of “make all”, or can be built standalone via “make bootloader-clean”. There is also “make bootloader-list-components” to see the components included in the bootloader build.

The component in IDF components/bootloader is special, as the second stage bootloader is a separate `.ELF` and `.BIN` file to the main project. However it shares its configuration and build directory with the main project.

This is accomplished by adding a subproject under `components/bootloader/subproject`. This subproject has its own Makefile, but it expects to be called from the project's own Makefile via some glue in the `components/bootloader/Makefile.projectbuild` file. See these files for more details.

5.6 Console

ESP-IDF provides `console` component, which includes building blocks needed to develop an interactive console over serial port. This component includes following facilities:

- Line editing, provided by `linenoise` library. This includes handling of backspace and arrow keys, scrolling through command history, command auto-completion, and argument hints.
- Splitting of command line into arguments.
- Argument parsing, provided by `argtable3` library. This library includes APIs useful for parsing GNU style command line arguments.
- Functions for registration and dispatching of commands.

These facilities can be used together or independently. For example, it is possible to use line editing and command registration features, but use `getopt` or custom code for argument parsing, instead of `argtable3`. Likewise, it is

possible to use simpler means of command input (such as `fgets`) together with the rest of the means for command splitting and argument parsing.

5.6.1 Line editing

Line editing feature lets users compose commands by typing them, erasing symbols using ‘backspace’ key, navigating within the command using left/right keys, navigating to previously typed commands using up/down keys, and performing autocompletion using ‘tab’ key.

Note: This feature relies on ANSI escape sequence support in the terminal application. As such, serial monitors which display raw UART data can not be used together with the line editing library. If you see `[\6n` or similar escape sequence when running `get_started/console` example instead of a command prompt (`[esp32]>`), it means that the serial monitor does not support escape sequences. Programs which are known to work are GNU screen, minicom, and `idf_monitor.py` (which can be invoked using `idf.py monitor` from project directory).

Here is an overview of functions provided by [linenoise](#) library.

Configuration

Linenoise library does not need explicit initialization. However, some configuration defaults may need to be changed before invoking the main line editing function.

linenoiseClearScreen Clear terminal screen using an escape sequence and position the cursor at the top left corner.

linenoiseSetMultiLine Switch between single line and multi line editing modes. In single line mode, if the length of the command exceeds the width of the terminal, the command text is scrolled within the line to show the end of the text. In this case the beginning of the text is hidden. Single line needs less data to be sent to refresh screen on each key press, so exhibits less glitching compared to the multi line mode. On the flip side, editing commands and copying command text from terminal in single line mode is harder. Default is single line mode.

Main loop

linenoise In most cases, console applications have some form of read/eval loop. `linenoise` is the single function which handles user’s key presses and returns completed line once ‘enter’ key is pressed. As such, it handles the ‘read’ part of the loop.

linenoiseFree This function must be called to release the command line buffer obtained from `linenoise` function.

Hints and completions

linenoiseSetCompletionCallback When user presses ‘tab’ key, `linenoise` library invokes completion callback. The callback should inspect the contents of the command typed so far and provide a list of possible completions using calls to `linenoiseAddCompletion` function. `linenoiseSetCompletionCallback` function should be called to register this completion callback, if completion feature is desired. `console` component provides a ready made function to provide completions for registered commands, `esp_console_get_completion` (see below).

linenoiseAddCompletion Function to be called by completion callback to inform the library about possible completions of the currently typed command.

linenoiseSetHintsCallback Whenever user input changes, `linenoise` invokes hints callback. This callback can inspect the command line typed so far, and provide a string with hints (which can include list of command arguments, for example). The library then displays the hint text on the same line where editing happens, possibly with a different color.

linenoiseSetFreeHintsCallback If hint string returned by hints callback is dynamically allocated or needs to be otherwise recycled, the function which performs such cleanup should be registered via `linenoiseSetFreeHintsCallback`.

History

linenoiseHistorySetMaxLen This function sets the number of most recently typed commands to be kept in memory. Users can navigate the history using up/down arrows.

linenoiseHistoryAdd Linenoise does not automatically add commands to history. Instead, applications need to call this function to add command strings to the history.

linenoiseHistorySave Function saves command history from RAM to a text file, for example on an SD card or on a filesystem in flash memory.

linenoiseHistoryLoad Counterpart to `linenoiseHistorySave`, loads history from a file.

linenoiseHistoryFree Releases memory used to store command history. Call this function when done working with linenoise library.

5.6.2 Splitting of command line into arguments

`console` component provides `esp_console_split_argv` function to split command line string into arguments. The function returns the number of arguments found (`argc`) and fills an array of pointers which can be passed as `argv` argument to any function which accepts arguments in `argc, argv` format.

The command line is split into arguments according to the following rules:

- Arguments are separated by spaces
- If spaces within arguments are required, they can be escaped using `\` (backslash) character.
- Other escape sequences which are recognized are `\\` (which produces literal backslash) and `\"`, which produces a double quote.
- Arguments can be quoted using double quotes. Quotes may appear only in the beginning and at the end of the argument. Quotes within the argument must be escaped as mentioned above. Quotes surrounding the argument are stripped by `esp_console_split_argv` function.

Examples:

- `abc def 1 20 .3` → `[abc, def, 1, 20, .3]`
- `abc "123 456" def` → `[abc, 123 456, def]`
- ``a\ b\\c\"` → `[a b\c"]`

5.6.3 Argument parsing

For argument parsing, `console` component includes `argtable3` library. Please see [tutorial](#) for an introduction to `argtable3`. Github repository also includes [examples](#).

5.6.4 Command registration and dispatching

`console` component includes utility functions which handle registration of commands, matching commands typed by the user to registered ones, and calling these commands with the arguments given on the command line.

Application first initializes command registration module using a call to `esp_console_init`, and calls `esp_console_cmd_register` function to register command handlers.

For each command, application provides the following information (in the form of `esp_console_cmd_t` structure):

- Command name (string without spaces)
- Help text explaining what the command does

- Optional hint text listing the arguments of the command. If application uses `Argtable3` for argument parsing, hint text can be generated automatically by providing a pointer to `argtable` argument definitions structure instead.
- The command handler function.

A few other functions are provided by the command registration module:

`esp_console_run` This function takes the command line string, splits it into `argc/argv` argument list using `esp_console_split_argv`, looks up the command in the list of registered components, and if it is found, executes its handler.

`esp_console_register_help_command` Adds `help` command to the list of registered commands. This command prints the list of all the registered commands, along with their arguments and help texts.

`esp_console_get_completion` Callback function to be used with `linenoiseSetCompletionCallback` from `linenoise` library. Provides completions to `linenoise` based on the list of registered commands.

`esp_console_get_hint` Callback function to be used with `linenoiseSetHintsCallback` from `linenoise` library. Provides argument hints for registered commands to `linenoise`.

5.6.5 Example

Example application illustrating usage of the `console` component is available in `examples/system/console` directory. This example shows how to initialize UART and VFS functions, set up `linenoise` library, read and handle commands from UART, and store command history in Flash. See `README.md` in the example directory for more details.

5.7 Deep Sleep Wake Stubs

ESP32 supports running a “deep sleep wake stub” when coming out of deep sleep. This function runs immediately as soon as the chip wakes up - before any normal initialisation, bootloader, or ESP-IDF code has run. After the wake stub runs, the SoC can go back to sleep or continue to start ESP-IDF normally.

Deep sleep wake stub code is loaded into “RTC Fast Memory” and any data which it uses must also be loaded into RTC memory. RTC memory regions hold their contents during deep sleep.

5.7.1 Rules for Wake Stubs

Wake stub code must be carefully written:

- As the SoC has freshly woken from sleep, most of the peripherals are in reset states. The SPI flash is unmapped.
- The wake stub code can only call functions implemented in ROM or loaded into RTC Fast Memory (see below.)
- The wake stub code can only access data loaded in RTC memory. All other RAM will be uninitialised and have random contents. The wake stub can use other RAM for temporary storage, but the contents will be overwritten when the SoC goes back to sleep or starts ESP-IDF.
- RTC memory must include any read-only data (`.rodata`) used by the stub.
- Data in RTC memory is initialised whenever the SoC restarts, except when waking from deep sleep. When waking from deep sleep, the values which were present before going to sleep are kept.
- Wake stub code is a part of the main `esp-idf` app. During normal running of `esp-idf`, functions can call the wake stub functions or access RTC memory. It is as if these were regular parts of the app.

5.7.2 Implementing A Stub

The wake stub in `esp-idf` is called `esp_wake_deep_sleep()`. This function runs whenever the SoC wakes from deep sleep. There is a default version of this function provided in `esp-idf`, but the default function is weak-linked so if your app contains a function named `esp_wake_deep_sleep()` then this will override the default.

If supplying a custom wake stub, the first thing it does should be to call `esp_default_wake_deep_sleep()`.

It is not necessary to implement `esp_wake_deep_sleep()` in your app in order to use deep sleep. It is only necessary if you want to have special behaviour immediately on wake.

If you want to swap between different deep sleep stubs at runtime, it is also possible to do this by calling the `esp_set_deep_sleep_wake_stub()` function. This is not necessary if you only use the default `esp_wake_deep_sleep()` function.

All of these functions are declared in the `esp_deeptime.h` header under `components/esp32`.

5.7.3 Loading Code Into RTC Memory

Wake stub code must be resident in RTC Fast Memory. This can be done in one of two ways.

The first way is to use the `RTC_IRAM_ATTR` attribute to place a function into RTC memory:

```
void RTC_IRAM_ATTR esp_wake_deep_sleep(void) {
    esp_default_wake_deep_sleep();
    // Add additional functionality here
}
```

The second way is to place the function into any source file whose name starts with `rtc_wake_stub`. Files names `rtc_wake_stub*` have their contents automatically put into RTC memory by the linker.

The first way is simpler for very short and simple code, or for source files where you want to mix “normal” and “RTC” code. The second way is simpler when you want to write longer pieces of code for RTC memory.

5.7.4 Loading Data Into RTC Memory

Data used by stub code must be resident in RTC memory. The data can be placed in RTC Fast memory or in RTC Slow memory which is also used by the ULP.

Specifying this data can be done in one of two ways:

The first way is to use the `RTC_DATA_ATTR` and `RTC_RODATA_ATTR` to specify any data (writeable or read-only, respectively) which should be loaded into RTC memory:

```
RTC_DATA_ATTR int wake_count;

void RTC_IRAM_ATTR esp_wake_deep_sleep(void) {
    esp_default_wake_deep_sleep();
    static RTC_RODATA_ATTR const char fmt_str[] = "Wake count %d\n";
    ets_printf(fmt_str, wake_count++);
}
```

The RTC memory area where this data will be placed can be configured via menuconfig option named `CONFIG_ESP32_RTCDATA_IN_FAST_MEM`. This option allows to keep slow memory area for ULP programs and once it is enabled the data marked with `RTC_DATA_ATTR` and `RTC_RODATA_ATTR` are placed in the RTC fast memory segment otherwise it goes to RTC slow memory (default option). This option depends on the `CONFIG_FREERTOS_UNICORE` because RTC fast memory can be accessed only by `PRO_CPU`.

The similar attributes `RTC_FAST_ATTR` and `RTC_SLOW_ATTR` can be used to specify data that will be force placed into `RTC_FAST` and `RTC_SLOW` memory respectively. Any access to data marked with `RTC_FAST_ATTR` is allowed by `PRO_CPU` only and it is responsibility of user to make sure about it.

Unfortunately, any string constants used in this way must be declared as arrays and marked with `RTC_RODATA_ATTR`, as shown in the example above.

The second way is to place the data into any source file whose name starts with `rtc_wake_stub`.

For example, the equivalent example in `rtc_wake_stub_counter.c`:

```
int wake_count;

void RTC_IRAM_ATTR esp_wake_deep_sleep(void) {
    esp_default_wake_deep_sleep();
    ets_printf("Wake count %d\n", wake_count++);
}
```

The second way is a better option if you need to use strings, or write other more complex code.

To reduce wake-up time use the `CONFIG_BOOTLOADER_SKIP_VALIDATE_IN_DEEP_SLEEP` Kconfig option, see more information in [Fast boot from Deep Sleep](#).

5.8 Error Handling

5.8.1 Overview

Identifying and handling run-time errors is important for developing robust applications. There can be multiple kinds of run-time errors:

- Recoverable errors:
 - Errors indicated by functions through return values (error codes)
 - C++ exceptions, thrown using `throw` keyword
- Unrecoverable (fatal) errors:
 - Failed assertions (using `assert` macro and equivalent methods) and `abort()` calls.
 - CPU exceptions: access to protected regions of memory, illegal instruction, etc.
 - System level checks: watchdog timeout, cache access error, stack overflow, stack smashing, heap corruption, etc.

This guide explains ESP-IDF error handling mechanisms related to recoverable errors, and provides some common error handling patterns.

For instructions on diagnosing unrecoverable errors, see [Fatal Errors](#).

5.8.2 Error codes

The majority of ESP-IDF-specific functions use `esp_err_t` type to return error codes. `esp_err_t` is a signed integer type. Success (no error) is indicated with `ESP_OK` code, which is defined as zero.

Various ESP-IDF header files define possible error codes using preprocessor defines. Usually these defines start with `ESP_ERR_` prefix. Common error codes for generic failures (out of memory, timeout, invalid argument, etc.) are defined in `esp_err.h` file. Various components in ESP-IDF may define additional error codes for specific situations.

For the complete list of error codes, see [Error Code Reference](#).

5.8.3 Converting error codes to error messages

For each error code defined in ESP-IDF components, `esp_err_t` value can be converted to an error code name using `esp_err_to_name()` or `esp_err_to_name_r()` functions. For example, passing `0x101` to `esp_err_to_name()` will return “ESP_ERR_NO_MEM” string. Such strings can be used in log output to make it easier to understand which error has happened.

Additionally, `esp_err_to_name_r()` function will attempt to interpret the error code as a [standard POSIX error code](#), if no matching `ESP_ERR_` value is found. This is done using `strerror_r` function. POSIX error codes (such as `ENOENT`, `ENOMEM`) are defined in `errno.h` and are typically obtained from `errno` variable. In ESP-IDF this variable is thread-local: multiple FreeRTOS tasks have their own copies of `errno`. Functions which set `errno` only modify its value for the task they run in.

This feature is enabled by default, but can be disabled to reduce application binary size. See [CONFIG_ESP_ERR_TO_NAME_LOOKUP](#). When this feature is disabled, `esp_err_to_name()` and `esp_err_to_name_r()` are still defined and can be called. In this case, `esp_err_to_name()` will return UNKNOWN_ERROR, and `esp_err_to_name_r()` will return Unknown error 0xXXXX(YYYYY), where 0xXXXX and YYYYY are the hexadecimal and decimal representations of the error code, respectively.

5.8.4 ESP_ERROR_CHECK macro

`ESP_ERROR_CHECK()` macro serves similar purpose as `assert`, except that it checks `esp_err_t` value rather than a `bool` condition. If the argument of `ESP_ERROR_CHECK()` is not equal `ESP_OK`, then an error message is printed on the console, and `abort()` is called.

Error message will typically look like this:

```
ESP_ERROR_CHECK failed: esp_err_t 0x107 (ESP_ERR_TIMEOUT) at 0x400d1fdf
file: "/Users/user/esp/example/main/main.c" line 20
func: app_main
expression: sdmmc_card_init(host, &card)

Backtrace: 0x40086e7c:0x3ffb4ff0 0x40087328:0x3ffb5010 0x400d1fdf:0x3ffb5030_
↳0x400d0816:0x3ffb5050
```

Note: If *IDF monitor* is used, addresses in the backtrace will be converted to file names and line numbers.

- The first line mentions the error code as a hexadecimal value, and the identifier used for this error in source code. The latter depends on [CONFIG_ESP_ERR_TO_NAME_LOOKUP](#) option being set. Address in the program where error has occurred is printed as well.
- Subsequent lines show the location in the program where `ESP_ERROR_CHECK()` macro was called, and the expression which was passed to the macro as an argument.
- Finally, backtrace is printed. This is part of panic handler output common to all fatal errors. See [Fatal Errors](#) for more information about the backtrace.

5.8.5 Error handling patterns

1. Attempt to recover. Depending on the situation, this might mean to retry the call after some time, or attempt to de-initialize the driver and re-initialize it again, or fix the error condition using an out-of-band mechanism (e.g reset an external peripheral which is not responding).

Example:

```
esp_err_t err;
do {
    err = sdio_slave_send_queue(addr, len, arg, timeout);
    // keep retrying while the sending queue is full
} while (err == ESP_ERR_TIMEOUT);
if (err != ESP_OK) {
    // handle other errors
}
```

2. Propagate the error to the caller. In some middleware components this means that a function must exit with the same error code, making sure any resource allocations are rolled back.

Example:

```
sdmmc_card_t* card = calloc(1, sizeof(sdmmc_card_t));
if (card == NULL) {
    return ESP_ERR_NO_MEM;
}
```

(continues on next page)

(continued from previous page)

```
esp_err_t err = sdmmc_card_init(host, &card);
if (err != ESP_OK) {
    // Clean up
    free(card);
    // Propagate the error to the upper layer (e.g. to notify the user).
    // Alternatively, application can define and return custom error code.
    return err;
}
```

3. Convert into unrecoverable error, for example using `ESP_ERROR_CHECK`. See [ESP_ERROR_CHECK macro](#) section for details.

Terminating the application in case of an error is usually undesirable behaviour for middleware components, but is sometimes acceptable at application level.

Many ESP-IDF examples use `ESP_ERROR_CHECK` to handle errors from various APIs. This is not the best practice for applications, and is done to make example code more concise.

Example:

```
ESP_ERROR_CHECK(spi_bus_initialize(host, bus_config, dma_chan));
```

5.8.6 C++ Exceptions

Support for C++ Exceptions in ESP-IDF is disabled by default, but can be enabled using [CONFIG_COMPILER_CXX_EXCEPTIONS](#) option.

Enabling exception handling normally increases application binary size by a few kB. Additionally it may be necessary to reserve some amount of RAM for exception emergency pool. Memory from this pool will be used if it is not possible to allocate exception object from the heap. Amount of memory in the emergency pool can be set using [CONFIG_COMPILER_CXX_EXCEPTIONS_EMG_POOL_SIZE](#) variable.

If an exception is thrown, but there is no `catch` block, the program will be terminated by `abort` function, and backtrace will be printed. See [Fatal Errors](#) for more information about backtraces.

See [system/cpp_exceptions](#) for an example of C++ exception handling.

5.9 ESP-BLE-MESH

Bluetooth® mesh networking enables many-to-many (m:m) device communications and is optimized for creating large-scale device networks.

Devices may relay data to other devices not in direct radio range of the originating device. In this way, mesh networks can span very large physical areas and contain large numbers of devices. It is ideally suited for building automation, sensor networks, and other IoT solutions where tens, hundreds, or thousands of devices need to reliably and securely communicate with one another.

Bluetooth mesh is not a wireless communications technology, but a networking technology. This technology is dependent upon Bluetooth Low Energy (BLE) - a wireless communications protocol stack.

Built on top of Zephyr Bluetooth Mesh stack, the ESP-BLE-MESH implementation supports device provisioning and node control. It also supports such node features as Proxy, Relay, Low power and Friend.

Please see the [ESP-BLE-MESH Architecture](#) for information about the implementation of ESP-BLE-MESH architecture and [ESP-BLE-MESH API Reference](#) for information about respective API.

ESP-BLE-MESH is implemented and certified based on the latest Mesh Profile v1.0.1, users can refer [here](#) for the certification details of ESP-BLE-MESH.

Note: If you are looking for Wi-Fi based implementation of mesh for ESP32, please check another product by Espressif called ESP-MESH. For more information and documentation see [ESP-MESH](#).

5.9.1 Getting Started with ESP-BLE-MESH

This section is intended to help you get started with ESP-BLE-MESH for the hardware based on the ESP32 chip by Espressif.

We are going to demonstrate process of setting and operation of a small ESP-BLE-MESH network of three nodes. This process will cover device provisioning and node configuration, and then sending on/off commands to Generic OnOff Server Models on specific nodes.

If you are new to ESP-IDF, please first set up development environment, compile, flash and run example application following top level ESP-IDF *Get Started* documentation.

What You Need

Hardware:

- Three ESP32 boards, see *options*.
- USB cables to connect the boards.
- Computer configured with ESP-IDF.
- Mobile phone or tablet running Android or iOS.

Software:

- Example application `bluetooth/esp_ble_mesh/ble_mesh_node/onoff_server` code to load to the ESP32 boards.
- Mobile App: **nRF Mesh** for Android or iOS. Optionally you can use some other Apps:
 - [EspBleMesh](#) Android App
 - Silicon Labs Android or iOS App

Installation Step by Step

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

Step 1. Check Hardware Both [ESP32-DevKitC](#) and [ESP-WROVER-KIT](#) development boards are supported for ESP-BLE-MESH implementation. You can choose particular board through menuconfig: `idf.py menuconfig`
> Example Configuration > Board selection for ESP-BLE-MESH

Note: If you plan to use [ESP32-DevKitC](#), connect a RGB LED to GPIO pins 25, 26 and 27.

Step 2. Configure Software Enter the `bluetooth/esp_ble_mesh/ble_mesh_node/onoff_server` example directory, run `idf.py menuconfig` to select your board and then run `idf.py build` to compile the example.

Step 3. Upload Application to Nodes After the `bluetooth/esp_ble_mesh/ble_mesh_node/onoff_server` example is compiled successfully, users can run `idf.py flash` to upload the same generated binary files into each of the three development boards.

Once boards are powered on, the RGB LED on each board should turn **GREEN**.

Step 4. Provision Nodes In this section, we will use the **nRF Mesh Android** App to demonstrate how to provision an unprovisioned device. Users can also get its iOS version from the App Store.

4.1 Scanner The Scanner is App's functionality to search for unprovisioned devices in range. Open the App, press **Scanner** at the bottom and the search will start. After a short while we should see three unprovisioned devices displayed.

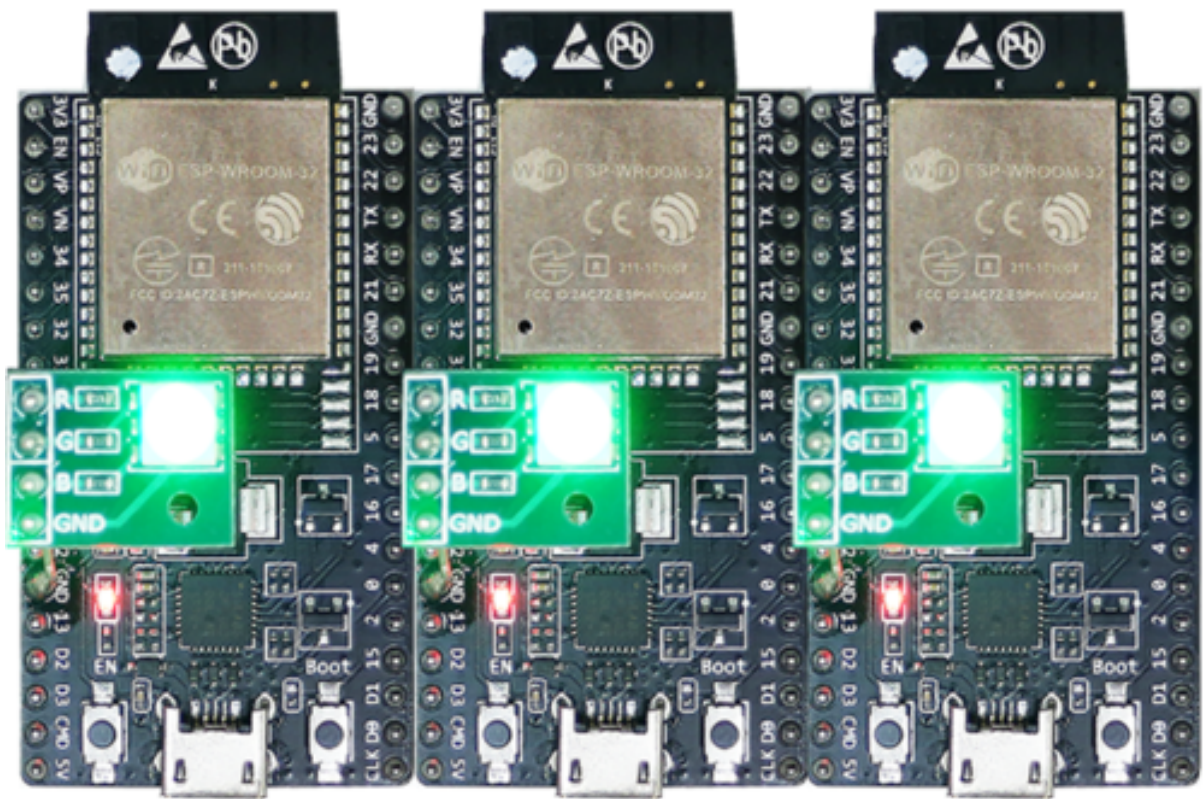


Fig. 4: ESP-BLE-MESH Devices Power On

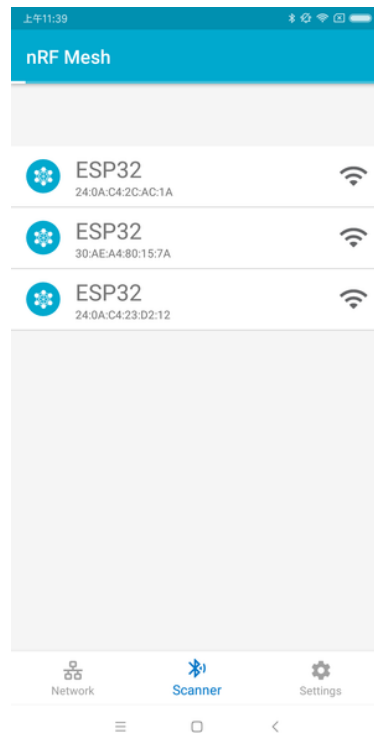


Fig. 5: nRF Mesh - Scanner

4.2 Identify Users can select any unprovisioned device, then the App will try to set up a connection with the selected device. After the BLE connection is established successfully (sometimes users need to try multiple times to get connected), and proper ESP-BLE-MESH GATT Service is discovered, users can see the **IDENTIFY** interface button on the screen. The IDENTIFY operation can be used to tell users which device is going to be provisioned.

Note: The IDENTIFY operation also needs some cooperation on the device side, then users can see which device is in the provisioning process. Currently when pressing the **IDENTIFY** interface button, no signs can be seen from the device except from the log on the serial monitor.

After the **IDENTIFY** interface button is pressed, users can see the **PROVISION** interface button.

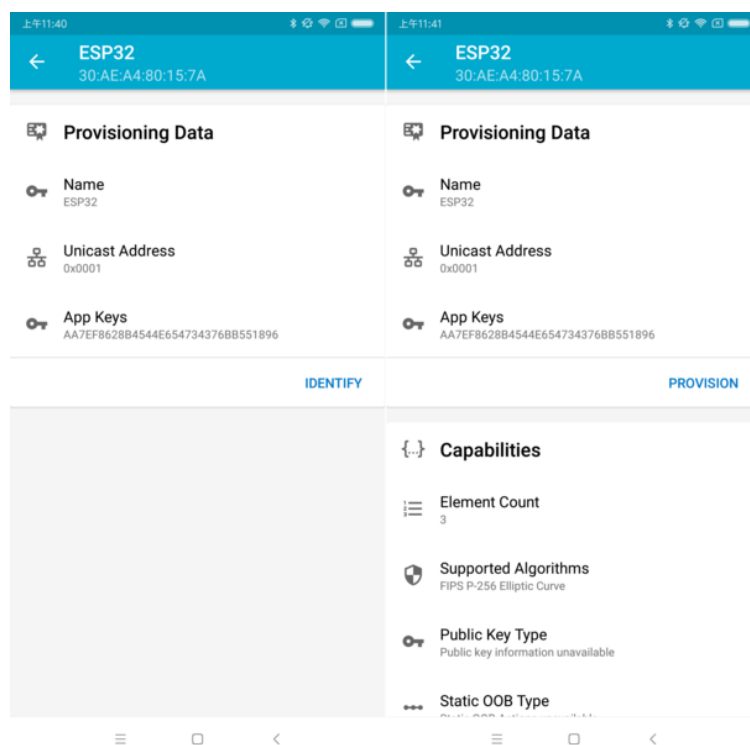


Fig. 6: nRF Mesh - IDENTIFY - PROVISION

4.3 Provision Then, the App will try to provision the unprovisioned device. When the device is provisioned successfully, the RGB LED on the board will turn off, and the App will implement the following procedures:

1. Disconnect with the node
2. Try to reconnect with the node
3. Connect successfully and discover ESP-BLE-MESH GATT Service
4. Get Composition Data of the node and add AppKey to it

When all the procedures are finished, the node is configured properly. And after pressing **OK**, users can see that unicast address is assigned, and Composition Data of the node is decoded successfully.

Sometimes in procedure 2, the App may fail to reconnect with the node. In this case, after pressing **OK**, users can see that only unicast address of the node has been assigned, but no Composition Data has been got. Then users need to press **CONNECT** on the top right, and the previously provisioned node will be displayed on the screen, and users need to choose it and try to connect with the node.

After connecting successfully, the App will show the interface buttons which can be used to get Composition Data and add AppKey.

If the device is the second or the third one which has been provisioned by the App, and after pressing **CONNECT**, users can see two or three nodes on the screen. In this situation, users can choose any device to connect with, once succeed then go back to the main screen to choose the node which needs to be configured.

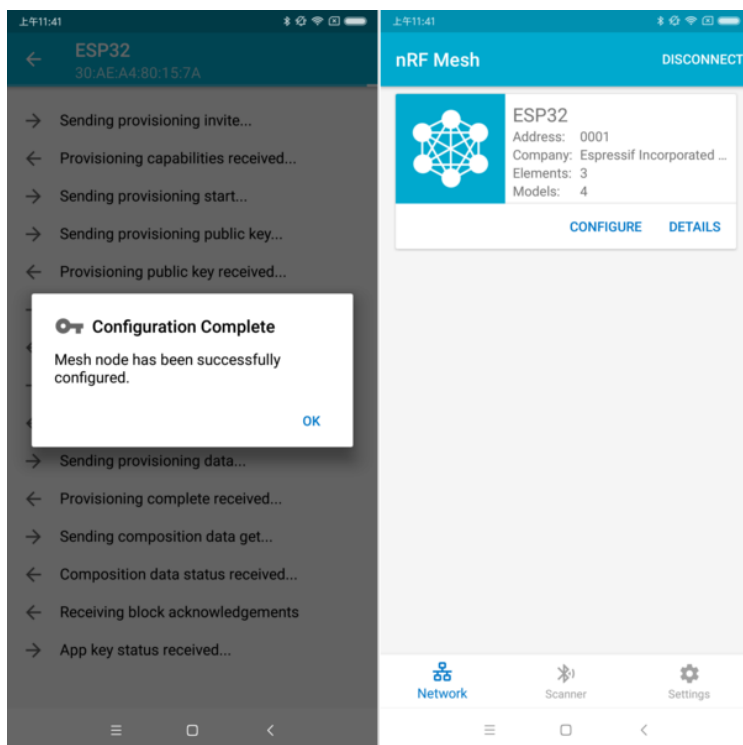


Fig. 7: nRF Mesh - Configuration Complete

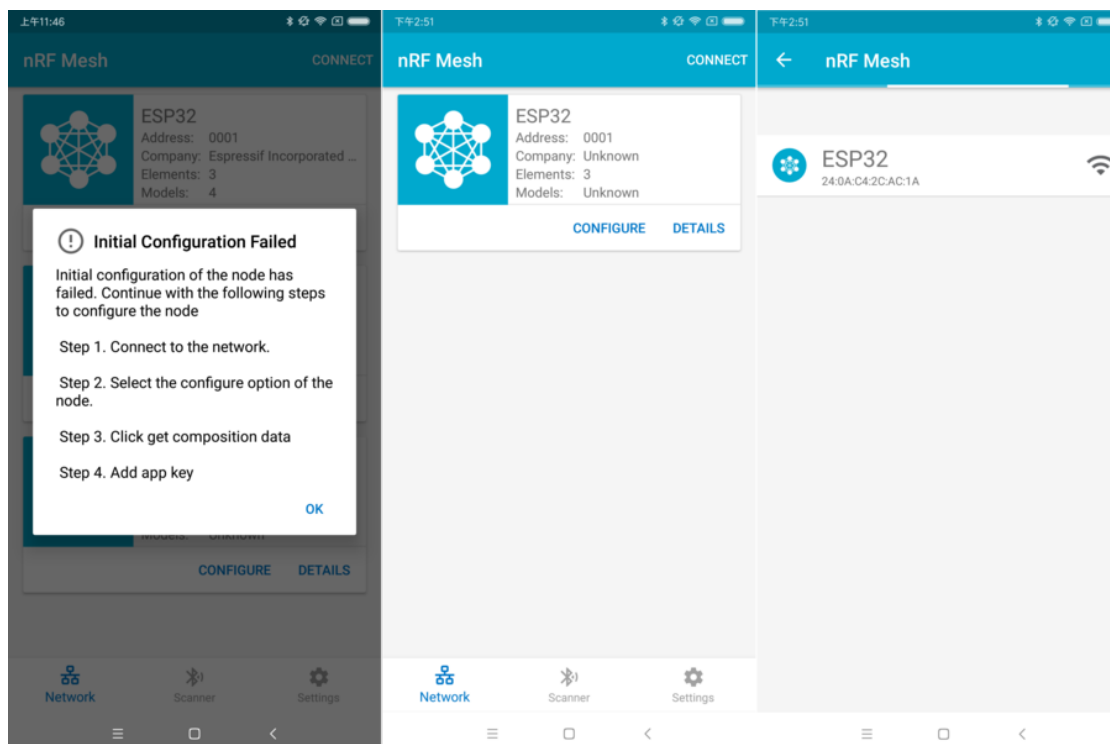


Fig. 8: nRF Mesh - Initial Configuration Failed

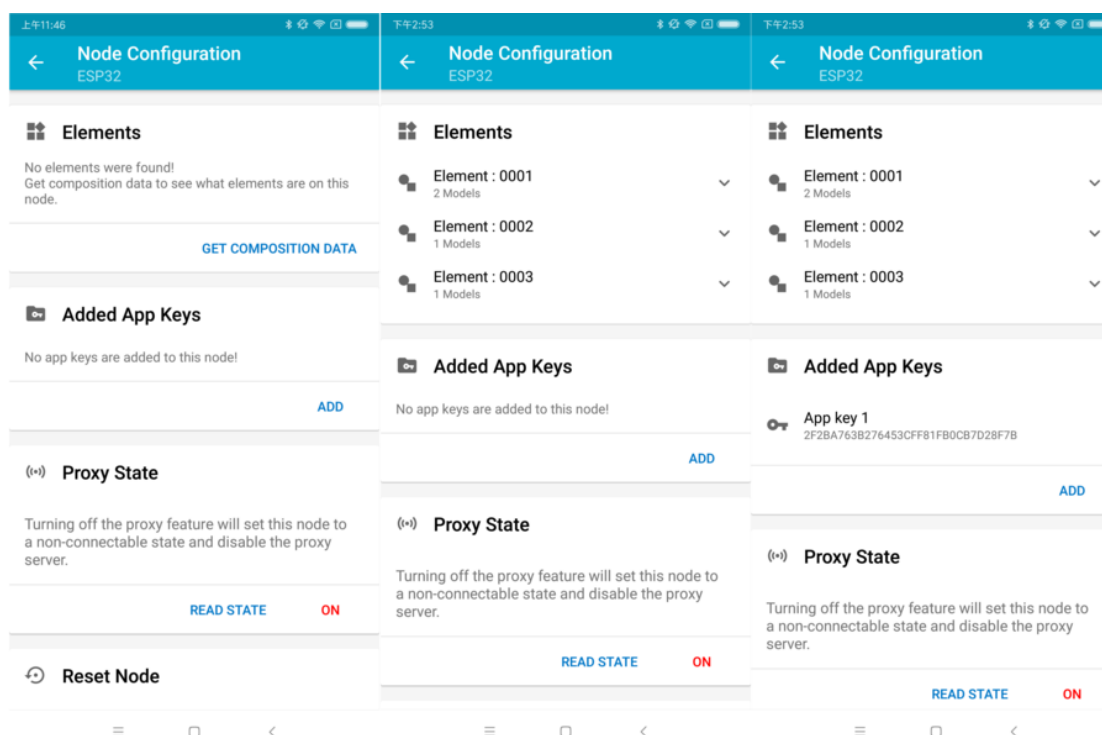


Fig. 9: nRF Mesh - Reconnect - Initial Configuration

Here an example of three devices listed.

- The left picture shows that the third device is provisioned successfully, but the App failed to connect with it. When it tries to reconnect with the third node, three nodes are displayed on the App.
- The right picture shows that after connecting with any node successfully, the App displays the information of the three nodes. Users can see that the App has got the Composition Data of the first and the second nodes, but for the third one, only the unicast address has been assigned to it while the Composition Data is unknown.

4.4 Configuration When provisioning and initial configuration are finished, users can start to configure the node, such as binding AppKey with each model with the elements, setting publication information to it, etc.

Example below shows how to bind AppKey with Generic OnOff Server Model within the Primary Element.

Note: No need to bind AppKey with the Configuration Server Model, since it only uses the DevKey to encrypt messages in the Upper Transport Layer.

Step 5. Operate Network After all the Generic OnOff Server Models within the three elements are bound with proper AppKey, users can use the App to turn on/off the RGB LED.

In the `bluetooth/esp_ble_mesh/ble_mesh_node/onoff_server` example, the first Generic OnOff Server Model is used to control the **RED** color, the second one is used to control the **GREEN** color and the third one is used to control the **BLUE** color.

The following screenshot shows different board with different color on.

Note: For **nRF Mesh** iOS App [version 1.0.4], when the node contains more than one element, the App is not behaving correctly. If users try to turn on/off the second or the third Generic OnOff Server Model, the message sent by the App is destined to the first Generic OnOff Server Model within the Primary Element.

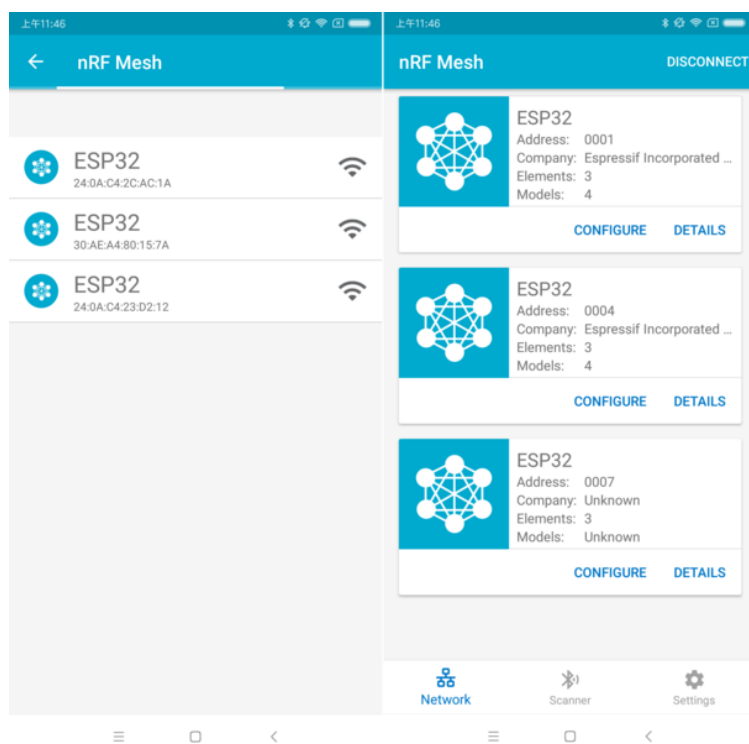


Fig. 10: nRF Mesh - Reconnect - Three Nodes

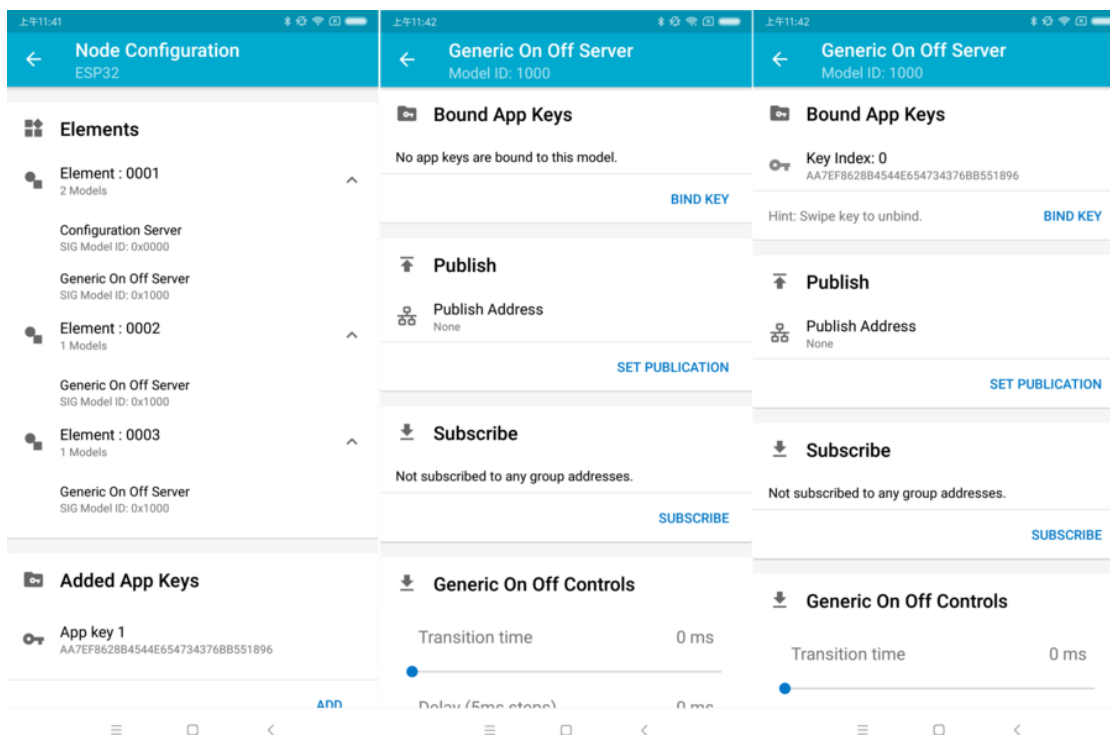


Fig. 11: nRF Mesh - Model Bind AppKey

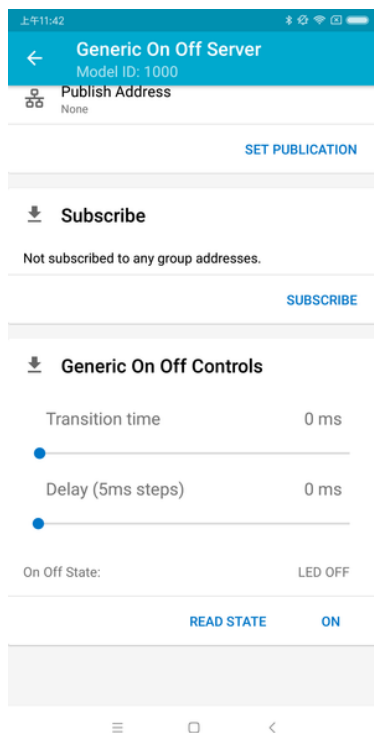


Fig. 12: nRF Mesh - Generic OnOff Control

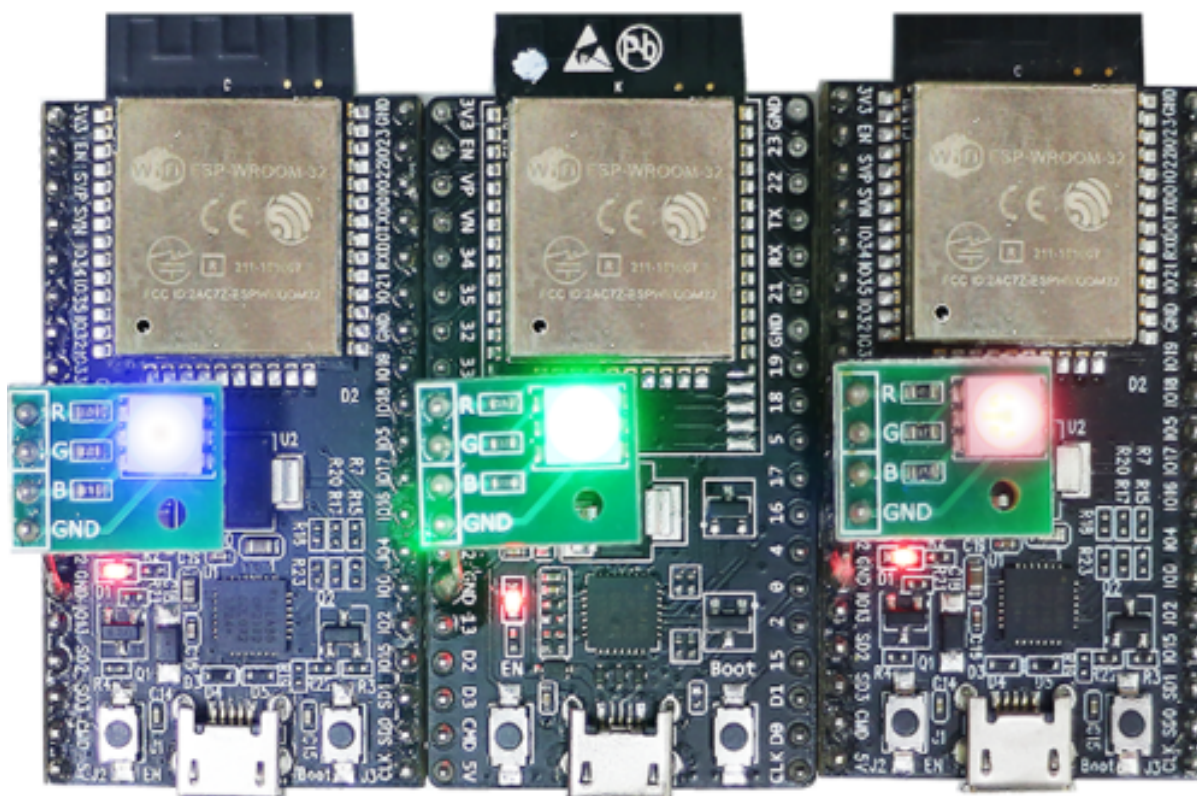


Fig. 13: Three ESP-BLE-MESH Nodes On

5.9.2 ESP-BLE-MESH Examples

- [ESP-BLE-MESH Node OnOff Server](#) - shows the use of ESP-BLE-MESH as a node having a Configuration Server model and a Generic OnOff Server model. A ESP-BLE-MESH Provisioner can then provision the unprovisioned device and control a RGB LED representing on/off state, see [example code](#) .
- [ESP-BLE-MESH Node OnOff Client](#) - shows how a Generic OnOff Client model works within a node. The node has a Configuration Server model and a Generic OnOff Client model, see [example code](#) .
- [ESP-BLE-MESH Provisioner](#) - shows how a device can act as an ESP-BLE-MESH Provisioner to provision devices. The Provisioner has a Configuration Server model, a Configuration Client model and a Generic OnOff Client model, see [example code](#) .
- [ESP-BLE-MESH Fast Provisioning - Client](#) and [Server](#) - this example is used for showing how fast provisioning can be used in order to create a mesh network. It takes no more than 60 seconds to provision 100 devices, see [example client code](#) and [example server code](#) .
- [ESP-BLE-MESH and Wi-Fi Coexistence](#) - an example that demonstrates the Wi-Fi and Bluetooth (BLE/BR/EDR) coexistence feature of ESP32. Simply put, users can use the Wi-Fi while operating Bluetooth, see [example code](#) .
- [ESP-BLE-MESH Node Console](#) - an example that implements BLE Mesh node basic features. Within this example a node can be scanned and provisioned by Provisioner and reply to get/set message from Provisioner, see [example node code](#) .

5.9.3 ESP-BLE-MESH Demo Videos

- [Provisioning of ESP-BLE-MESH nodes using Smartphone App](#)
- [Espressif Fast Provisioning using ESP-BLE-MESH App](#)
- [Espressif ESP-BLE-MESH and Wi-Fi Coexistence](#)

5.9.4 ESP-BLE-MESH FAQ

- [1. Provisioner Development](#)
- [2. Node Development](#)
- [3. ESP-BLE-MESH and Wi-Fi Coexistence](#)
- [4. Fast Provisioning](#)
- [5. Log Help](#)
- [6. Example Help](#)
- [7. Others](#)

5.9.5 Related Documents

ESP-BLE-MESH Feature List

Supported Features

Mesh Core

- **Provisioning: Node Role**
 - PB-ADV and PB-GATT
 - OOB Authentication
- **Provisioning: Provisioner Role**
 - PB-ADV and PB-GATT
 - OOB Authentication
- **Networking**
 - Relay
 - Segmentation and Reassembly
 - Key Refresh Procedure
 - IV Update Procedure

- Friend
- Low Power
- Proxy Server
- Proxy Client
- **Multiple Client Models Run Simultaneously**
 - Support multiple client models send packets to different nodes simultaneously
 - No blocking between client model and server model
- **NVS Storing**
 - Store provisioning and configuration information of ESP-BLE-MESH Node

Mesh Models

- **Foundation models**
 - Configuration Server model
 - Configuration Client model
 - Health Server model
 - Health Client model
- **Generic client models**
 - Generic OnOff Client
 - Generic Level Client
 - Generic Default Transition Time Client
 - Generic Power OnOff Client
 - Generic Power Level Client
 - Generic Battery Client
 - Generic Location Client
 - Generic Property Client
- **Sensor client models**
 - Sensor Client
- **Time and Scenes client models**
 - Time Client
 - Scene Client
 - Scheduler Client
- **Lighting client models**
 - Light Lightness Client
 - Light CTL Client
 - Light HSL Client
 - Light xyL Client
 - Light LC Client
- **Generic server models**
 - Generic OnOff Server
 - Generic Level Server
 - Generic Default Transition Time Server
 - Generic Power OnOff Server
 - Generic Power OnOff Setup Server
 - Generic Power Level Server
 - Generic Power Level Setup Server
 - Generic Battery Server
 - Generic Location Server
 - Generic Location Setup Server
 - Generic User Property Server
 - Generic Admin Property Server
 - Generic Manufacturer Property Server
 - Generic Client Property Server
- **Sensor server models**
 - Sensor Server
 - Sensor Setup Server
- **Time and Scenes server models**
 - Time Server

- Time Setup Server
- Scene Server
- Scene Setup Server
- Scheduler Server
- Scheduler Setup Server
- **Lighting server models**
 - Light Lightness Server
 - Light Lightness Setup Server
 - Light CTL Server
 - Light CTL Temperature Server
 - Light CTL Setup Server
 - Light HSL Server
 - Light HSL Hue Server
 - Light HSL Saturation Server
 - Light HSL Setup Server
 - Light xyL Server
 - Light xyL Setup Server
 - Light LC Server
 - Light LC Setup Server

Mesh Applications

- **ESP-BLE-MESH Node**
 - [Tutorial](#)
 - [Tutorial](#)
 - [Example](#)
- **ESP-BLE-MESH Provisioner**
 - [Tutorial](#)
 - [Example](#)
- **ESP-BLE-MESH Fast Provisioning**
 - [Fast Provisioning Client Model Tutorial](#)
 - [Fast Provisioning Server Model Tutorial](#)
 - [Example](#)
 - [Demo Video](#)
- **ESP-BLE-MESH and Wi-Fi Coexistence**
 - [Tutorial](#)
 - [Example](#)
 - [Demo Video](#)
- **ESP-BLE-MESH Console Commands**
 - [Example](#)

Future Release Features

Mesh Core

- Provisioner NVS Storage

Mesh Applications

- Fast OTA
- Friendship

ESP-BLE-MESH Architecture

This document introduces ESP-BLE-MESH architecture overview, ESP-BLE-MESH architecture implementation as well as ESP-BLE-MESH auxiliary routines.

- ESP-BLE-MESH Architecture Overview
 - Describes the five major parts of ESP-BLE-MESH architecture and the functionality of each part.
- ESP-BLE-MESH Architecture Implementation
 - Describes the basic functions of ESP-BLE-MESH files, the correspondence between files and ESP-BLE-MESH architecture, and the interface for calling among files.
- ESP-BLE-MESH Auxiliary Routines
 - Describe the auxiliary routines of ESP-BLE-MESH, such as Mesh network management, Mesh features, etc.

1. ESP-BLE-MESH Architecture Overview Currently ESP-BLE-MESH has implemented most functions of Mesh Profile and all the Client Models defined in Mesh Model specification. Those missing functions/models are under development and will be provided soon. ESP-BLE-MESH architecture has been granted the official Bluetooth certification.

ESP-BLE-MESH architecture includes five key parts:

- Mesh Protocol Stack
 - Mesh Networking is responsible for processing of messages of ESP-BLE-MESH nodes.
 - Mesh Provisioning is responsible for provisioning flow of ESP-BLE-MESH devices.
 - Mesh Models is responsible for the implementation of SIG-defined models.
- Network Management
 - Implements several network management procedures, including node removal procedure, IV Index recovery procedure, etc.
- Features
 - Include several ESP-BLE-MESH features, e.g. Low Power feature, Friend feature, Relay feature, etc.
- Mesh Bearer Layer
 - Includes Advertising Bearer and GATT Bearer. The bearer layer is crucial to ESP-BLE-MESH protocol stack which is built on Bluetooth Low-Energy technology, because the protocol stack must make use of the bearer layer to transmit data via the BLE advertising channel and connection channel.
- Applications
 - Based on ESP-BLE-MESH protocol stack and Mesh Models.
 - By calling API and handling Event, Applications interact with Mesh Networking and Mesh Provisioning in ESP-BLE-MESH protocol stack, as well as a series of Models provided by Mesh Models.

1.1 Mesh Protocol Stack

1.1.1 Mesh Networking Mesh Networking in the protocol stack architecture implements the following functions:

- The communication between nodes in the Mesh network.
- Encryption and decryption of messages in the Mesh network.
- Management of Mesh network resources (Network Key, IV Index, etc.).
- Segmentation and reassembly of Mesh network messages.
- Model mapping of messages between different models.
- For more features, please see [ESP-BLE-MESH Feature List](#).

The implementation of Mesh Networking functions is based on hierarchy structure. Functions of each layer are shown in Table 1.1:

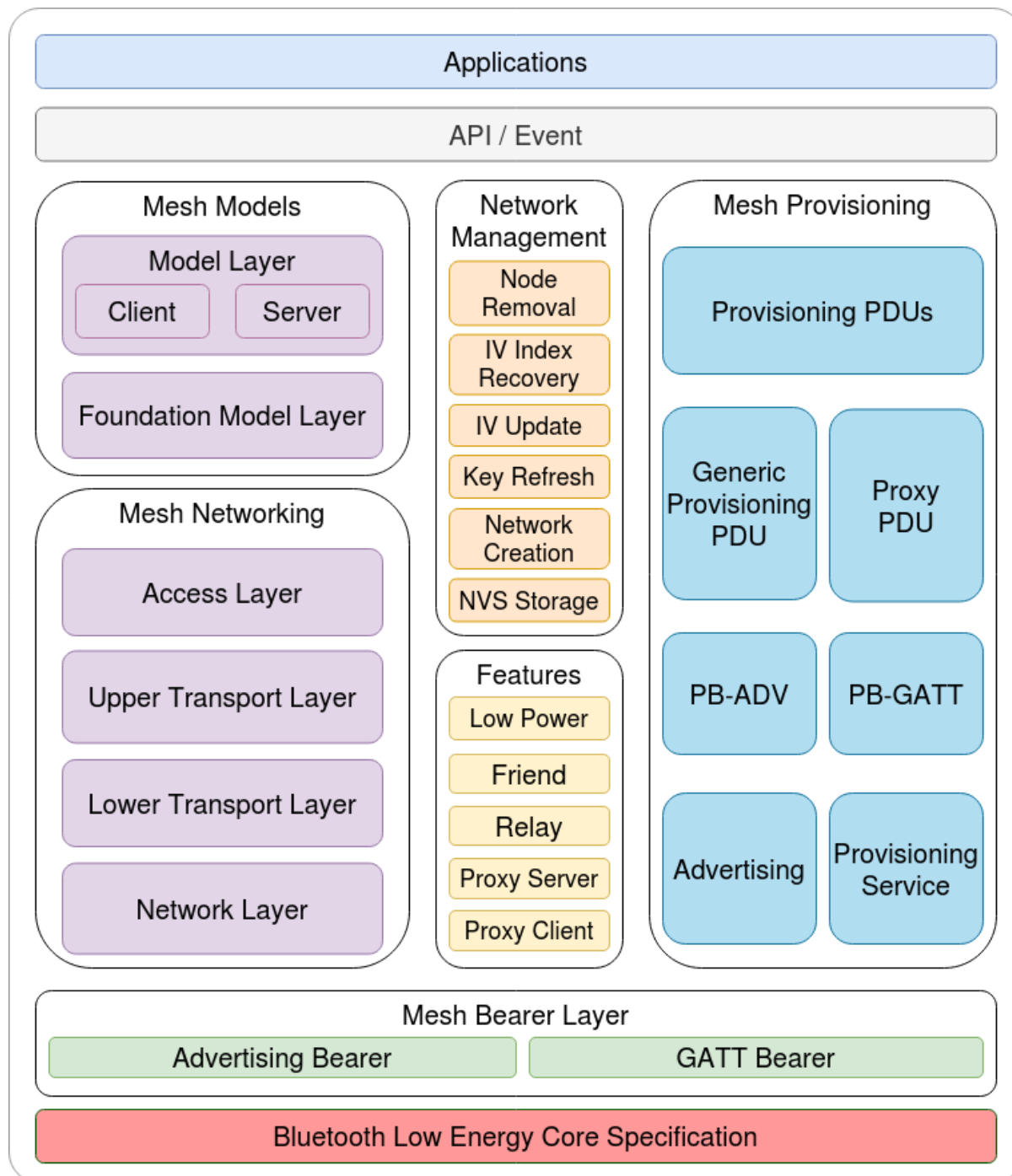


Fig. 14: Figure 1.1 ESP-BLE-MESH Architecture Diagram

Table 2: Table 1.1 Mesh Networking Architecture Description

| Layer | Function |
|-----------------------|--|
| Access Layer | Access Layer not only defines the format of application data, but also defines and controls the encryption and decryption of the data packets conducted by Upper Transport Layer. |
| Upper Transport Layer | Upper Transport Layer encrypts, decrypts, and authenticates application data to and from the access layer; it also handles special messages called “transport control messages” , including messages related to “friendship” and heartbeat messages. |
| Lower Transport Layer | Lower Transport Layer handles segmentation and reassembly of PDU. |
| Network Layer | Network Layer defines the address type and format of the network messages, and implements the relay function of the device. |

1.1.2 Mesh Provisioning Mesh Provisioning in the protocol stack architecture implements the following functions:

- Provisioning of unprovisioned devices.
- Allocation of Mesh network resources (unicast address, IV Index, NetKey, etc.).
- Four authentication methods support during provisioning.
- For more features, please see [ESP-BLE-MESH Feature List](#).

The implementation of Mesh Provisioning functions is based on hierarchy structure. Functions of each layer are shown in Table 1.2:

Table 3: Table 1.2 Mesh Provisioning Architecture Description

| Layer | Function |
|------------------------------------|--|
| Provisioning PDUs | Provisioning PDUs from different layers are handled using provisioning protocol. |
| Generic Provisioning PDU/Proxy PDU | The Provisioning PDUs are transmitted to an unprovisioned device using a Generic Provisioning layer or Proxy protocol layer. |
| PB-ADV/PB-GATT | These layers define how the Provisioning PDUs are transmitted as transactions that can be segmented and reassembled. |
| Advertising/Provisioning Service | The provisioning bearers define how sessions are established such that the transactions from the generic provisioning layer can be delivered to a single device. |

1.1.3 Mesh Models Mesh Models in the protocol stack architecture implements the following functions:

- Configuration Client/Server Models
- Health Client/Server Models
- Generic Client/Server Models
- Sensor Client/Server Models
- Time and Scenes Client/Server Models
- Lighting Client/Server Models

Functions of each layer are shown in Table 1.3:

Table 4: Table 1.3 Mesh Models Architecture Description

| Layer | Function |
|------------------------|--|
| Model Layer | Model Layer implements models used to standardize the operation of typical user scenarios, including Generic Client/Server Models, Sensor Client/Server Models, Time and Scenes Client/Server Models, Lighting Client/Server Models and several vendor models. |
| Foundation Model Layer | Foundation Model Layer implements models related to ESP-BLE-MESH configuration, management, self diagnosis, etc. |

1.2 Mesh Network Management Network Management implements the following functions:

- Node removal procedure is used to remove a node from the network.

- IV Index recovery procedure is used to recover a node's IV Index.
- IV update procedure is used to update the nodes' IV Index.
- Key refresh procedure is used to update the nodes' NetKey, AppKey, etc.
- Network creation procedure is used to create a mesh network.
- NVS storage is used to store node's networking information.

1.3 Mesh Features Features includes the following options:

- Low Power feature is used to reduce node's power consumption.
- Friend feature is used to store messages for Low Power nodes.
- Relay feature is used to relay/forward Network PDUs received by a node over the advertising bearer.
- Proxy Server/Client are two node roles in proxy protocol, which enable nodes to send and receive Network PDUs, mesh beacons, proxy configuration messages and Provisioning PDUs over a connection-oriented bearer.

1.4 Mesh Bearer Layer Bearers in the protocol stack architecture are responsible for passing of data between ESP-BLE-MESH protocol stack and Bluetooth Low Energy Core.

Bearers can be taken as a carrier layer based on Bluetooth Low Energy Core, which implements the function of receiving and transmitting data for the ESP-BLE-MESH protocol stack.

Table 5: Table 1.3 Mesh Bearers Description

| Layer | Function |
|--------------------|--|
| GATT Bearer | The GATT Bearer uses the Proxy protocol to transmit and receive Proxy PDUs between two devices over a GATT connection. |
| Advertising Bearer | When using the Advertising Bearer, a mesh packet shall be sent in the Advertising Data of a Bluetooth Low Energy advertising PDU using the Mesh Message AD Type. |

1.5 Mesh Applications The Applications in the protocol stack architecture implement the corresponding functions by calling the API provided by the ESP-BLE-MESH protocol stack and processing the Event reported by the protocol stack. There are some common applications, such as gateway, lighting and etc.

Interaction between application layer (Applications) and API / Event

- Application layer calls API
 - Call the provisioning-related API for provisioning.
 - Call the model-related API to send messages.
 - Call the device-attributes-related API to get local information about the device.
- Application layer processes Event

The application layer is designed based on events, which take parameters to the application layer. Events are mainly divided into two categories.

 - **The events completed by calling API.**
 - * Such as nodes sending messages.
 - **The events that the protocol stack actively reports to the application layer.**
 - * The Event that the protocol stack actively reports.
 - * The Event that Model actively reports.
- The event is reported by the callback function registered by the application layer, and the callback function also contains the corresponding processing of the event.

Interaction between API / Event and ESP-BLE-MESH protocol stack

- API used by user mainly calls functions provided by Mesh Networking, Mesh Provisioning and Mesh Models.
- The interaction between API / Event and the protocol stack does not operate across the hierarchy of the protocol stack. For example, API does not call functions related to Network Layer.

2. ESP-BLE-MESH Architecture Implementation The design and implementation of ESP-BLE-MESH architecture is based on layers and modules. In details, Section 2.1 (Mesh Networking Implementation), Section 2.2 (Mesh Provisioning Implementation) and Section 2.3 (Mesh Bearers Implementation) are based on layers, and Section 2.4 (Mesh Models Implementation) is on modules.

- **Layer-based Approach:** With Layer-based approach, the architecture is designed according to the layers specified in the Mesh Profile Specification. Each layer has its unique files which include APIs of this layer and etc. The specific design is shown in Figure 2.1.
- **Module-based Approach:** Every file implements an independent function that can be called by other programs.

The design of ESP-BLE-MESH architecture uses layer-based approach. The sequence of layers which data packets are processed through is fixed, i.e., the processing of packets will form a `message flow`. Thus, we could see flows of messages from the Protocol Stack Interface Diagram in Figure 2.1.

2.1 Mesh Protocol Stack Implementation

2.1.1 Mesh Networking Implementation The list of files and the functions implemented in each file in Mesh Networking are shown in Table 2.1:

Table 6: Table 2.1 Mesh Networking File Description :widths: 40 150
:header-rows: 1

| File | Functionality |
|------------------------------|--|
| ac-cess.c | ESP-BLE-MESH Access Layer |
| trans-port.c | ESP-BLE-MESH Lower/Upper Transport Layer |
| net.c | ESP-BLE-MESH Network Layer |
| adv.c | A task used to send ESP-BLE-MESH advertising packets, a callback used to handle received advertising packets and APIs used to allocate adv buffers |

2.1.2 Mesh Provisioning Implementation The implementation of Mesh Provisioning is divided into two chunks due to the Node/Provisioner coexistence.

Specific files that provide implementation of provisioning of Node are shown in Table 2.2:

Table 7: Table 2.2 Mesh Provisioning (Node) File Description

| File | Functionality |
|--------------------------------|---|
| prov.c | ESP-BLE-MESH Node provisioning (PB-ADV & PB-GATT) |
| proxy_server.c | ESP-BLE-MESH Proxy Server related functionalities |
| beacon.c | APIs used to handle ESP-BLE-MESH Beacons |

Specific files that implement functions of Provisioner are shown in Table 2.3:

Table 8: Table 2.3 Mesh Provisioning (Provisioner) File Description

| File | Functionality |
|------------------------------------|---|
| provisioner_prov.c | ESP-BLE-MESH Provisioner provisioning (PB-ADV & PB-GATT) |
| proxy_client.c | ESP-BLE-MESH Proxy Client related functionalities |
| provisioner_main.c | ESP-BLE-MESH Provisioner networking related functionalities |

2.1.3 Mesh Models Implementation Mesh Models are used to implement the specific functions of model in nodes. Server model is used to maintain node status. Client model is used to obtain and modify node state.

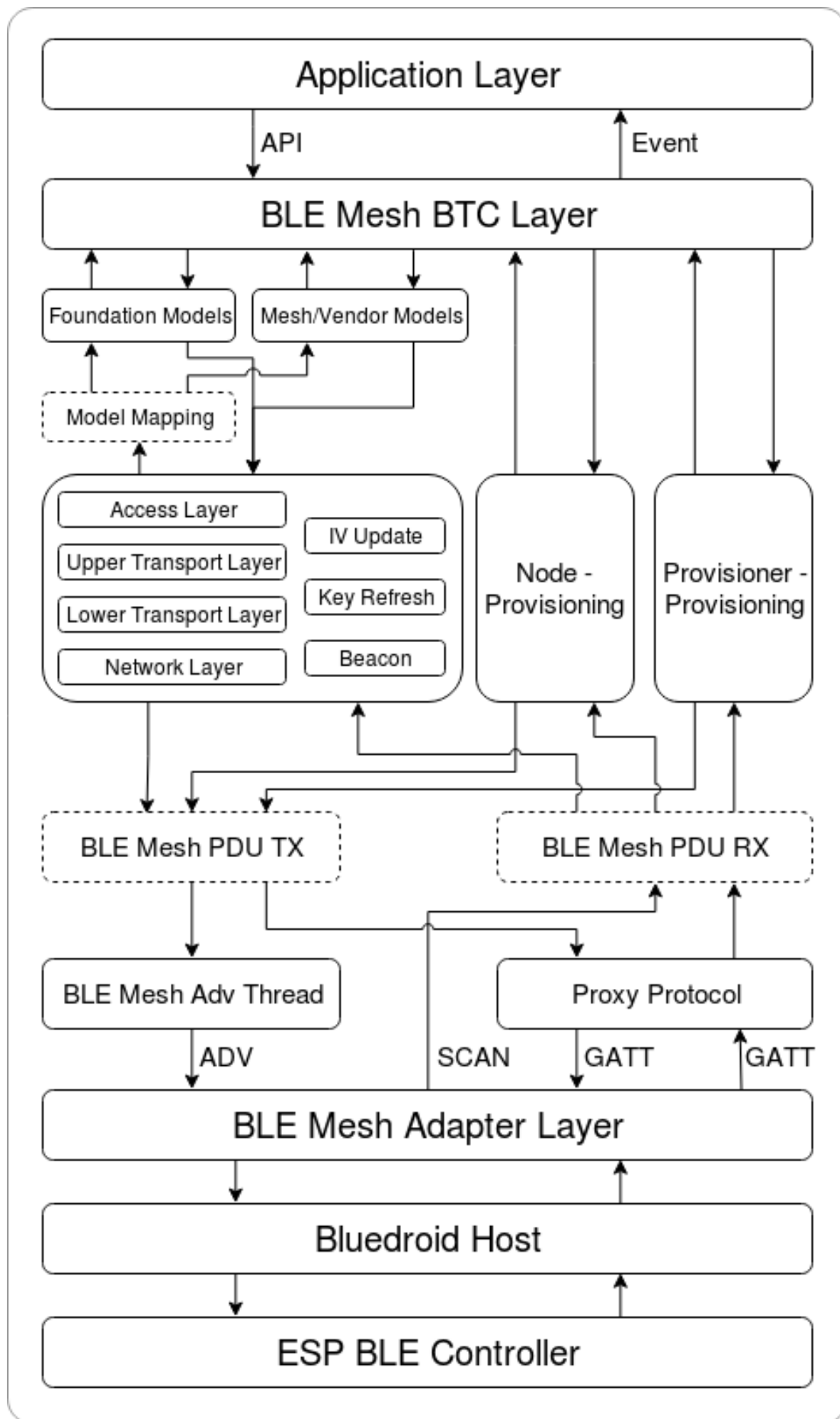


Fig. 15: Figure 2.1 ESP-BLE-MESH Architecture Implementation Diagram

Table 9: Table 2.4 Mesh Models File Description

| File | Functionality |
|-------------------------------------|--|
| cfg_cli.c | Send Configuration Client messages and receive corresponding response messages |
| cfg_srv.c | Receive Configuration Client messages and send proper response messages |
| health_cli.c | Send Health Client messages and receive corresponding response messages |
| health_srv.c | Receive Health Client messages and send proper response messages |
| client_common.c | ESP-BLE-MESH model related operations |
| generic_client.c | Send ESP-BLE-MESH Generic Client messages and receive corresponding response messages |
| lighting_client.c | Send ESP-BLE-MESH Lighting Client messages and receive corresponding response messages |
| sensor_client.c | Send ESP-BLE-MESH Sensor Client messages and receive corresponding response messages |
| time_scene_client.c | Send ESP-BLE-MESH Time Scene Client messages and receive corresponding response messages |
| generic_server.c | Receive ESP-BLE-MESH Generic Client messages and send corresponding response messages |
| lighting_server.c | Receive ESP-BLE-MESH Lighting Client messages and send corresponding response messages |
| sensor_server.c | Receive ESP-BLE-MESH Sensor Client messages and send corresponding response messages |
| time_scene_server.c | Receive ESP-BLE-MESH Time Scene Client messages and send corresponding response messages |

2.2 Mesh Bearers Implementation Portability is fully considered in the implementation of Mesh Bearers. When the ESP-BLE-MESH protocol stack is being ported to other platforms, users only need to modify [mesh_bearer_adapt.c](#) .

Table 10: Table 2.5 Mesh Bearers File Description

| File | Functionality |
|-------------------------------------|---|
| mesh_bearer_adapt.c | ESP-BLE-MESH Bearer Layer adapter , This file provides the interfaces used to receive and send ESP-BLE-MESH ADV & GATT related packets. |

Note: [mesh_bearer_adapt.c](#) is the implementation of Advertising Bearer and GATT Bearer in Mesh Networking framework.

2.3 Mesh Applications Implementation We have provided a series of application examples for customer development, and users can develop products based on [ESP-BLE-MESH Examples](#).

3. Auxiliary Routine Auxiliary routine refers to optional functions in the ESP-BLE-MESH protocol stack. The design of the auxiliary routine generally implement the truncation of code through [CONFIG_BLE_MESH](#).

3.1 Features

- Low Power
- Friend
- Relay
- Proxy Client/Server

3.2 Network Management

- Node Removal procedure
- IV Index Recovery procedure
- IV Update procedure
- Key Refresh procedure
- Network Creation procedure
- NVS Storage

3.3 Auxiliary Routine Implementation When adopting the design of independent module, the two main factors should be considered:

- The module can not be implemented hierarchically, and it can be completely independent, which means it does not rely on the implementation of other modules.
- The functions in the module will be used repeatedly, so it is reasonable to design it into a module. Independent module is shown in Table 3.1:

Table 11: Table 3.1 Module File Description

| File | Functionality |
|--------------------------------|---|
| lpn.c | ESP-BLE-MESH Low Power functionality |
| friend.c | ESP-BLE-MESH Friend functionality |
| net.c | ESP-BLE-MESH Relay feature, network creation, IV Update procedure, IV Index recovery procedure, Key Refresh procedure related functionalities |
| proxy_server.c | ESP-BLE-MESH Proxy Server related functionalities |
| proxy_client.c | ESP-BLE-MESH Proxy Client related functionalities |
| settings.c | ESP-BLE-MESH NVS storage functionality |
| main.c | ESP-BLE-MESH stack initialize, stack enable, node removal related functionalities |

ESP-BLE-MESH FAQ

This document provides a summary of frequently asked questions about developing with ESP-BLE-MESH, and is divided into seven sections:

- [1. Provisioner Development](#)
- [2. Node Development](#)
- [3. ESP-BLE-MESH and Wi-Fi Coexistence](#)
- [4. Fast Provisioning](#)
- [5. Log Help](#)
- [6. Example Help](#)
- [7. Others](#)

Users could refer to the sections for quick answer to their questions. This document will be updated based on the feedback collected via various channels.

1. Provisioner Development Generally, a Provisioner is used to provision unprovisioned devices and form a mesh network. And after provisioning, roles of the unprovisioned devices will be changed to those of a node.

1.1 What is the flow for an unprovisioned device to join ESP-BLE-MESH network?

There are two phases for a device to join ESP-BLE-MESH network via a Provisioner, namely, provisioning and configuration.

- The phase of provisioning is to assign unicast address, add NetKey and etc. to a device. By provisioning, the device joins the ESP-BLE-MESH network and its role is changed from an unprovisioned device to a node.

- The phase of configuration is to add AppKeys to the node and bind AppKeys to corresponding models. And some items are optional during configuration, including adding subscription addresses to the node, set publication information, etc. By configuration, the node can actually transmit messages to a Provisioner and receive messages from it.

1.2 If a Provisioner wants to change states of a node, what requirements should be met for a Provisioner?

- Client model that corresponds to server model of the node is required.
- NetKey and AppKey used to encrypt messages shall be owned by both the node and the Provisioner.
- The address owned by the node shall be known, which could be its unicast address or subscription address.

1.3 How can NetKey and AppKey be used?

- NetKey is used for encryption of messages in Network Layer. Nodes with the same NetKey are assumed to be in the same subnet while those with different NetKeys cannot communicate with each other.
- AppKey is used for encryption of messages in Upper Transport Layer. If client model and server model are bound to different AppKeys, the communication cannot be achieved.

1.4 How to generate a NetKey or AppKey for Provisioner? Can we use a fixed NetKey or AppKey?

- The API `esp_ble_mesh_provisioner_add_local_net_key()` can be used to add a NetKey with a fixed or random value.
- The API `esp_ble_mesh_provisioner_add_local_app_key()` can be used to add an AppKey with a fixed or random value.

1.5 Is the unicast address of Provisioner fixed?

The value of `prov_unicast_addr` in `esp_ble_mesh_prov_t` is used to set the unicast address of Provisioner, it can be set only once during initialization and can't be changed afterwards.

1.6 Can the address of Provisioner serve as destination address of the node-reporting-status message ?

The unicast address of Provisioner can be set only once during initialization and can't be changed afterwards. In theory, it can serve as the destination address of the node-reporting-status message, provided that the unicast address of the Provisioner is known by nodes. Nodes can know the unicast address of Provisioner during configuration since Provisioner sends messages to them with its unicast address used as the source address. Subscription address can also be used. Provisioner subscribes to a group address or virtual address, and nodes send messages to the subscription address.

1.7 Is the unicast address of the node that is firstly provisioned by Provisioner to ESP-BLE-MESH network fixed ?

The value of `prov_start_address` in `esp_ble_mesh_prov_t` is used to set the starting address when the Provisioner provisions unprovisioned devices, i.e. the unicast address of the node it firstly provisioned. It can be set only once during initialization and can't be changed afterwards.

1.8 Is the unicast address of the node that mobile App firstly provisioned fixed?

The App will decide the unicast address, and currently most of them are fixed.

1.9 How to know which unprovisioned device is the Provisioner that is provisioning currently?

The value of `prov_attention` in `esp_ble_mesh_prov_t` is used by Provisioner set to unprovisioned device during provisioning. It can be set only once during initialization and can't be changed afterwards. When the unprovisioned device is joining the mesh network, it can display in a specific way like flashing light to notify Provisioner that it is being provisioned.

1.10 How many ways to authenticate the devices during provisioning? Which way was used in the provided examples ?

There are four authentication methods, i.e. No OOB, Static OOB, Output OOB and Input OOB. In the provided examples, No OOB is used.

1.11 What information can be carried by the advertising packets of the unprovisioned device before provisioning into the network?

- Device UUID
- OOB Info
- URL Hash (optional)

1.12 Can such information be used for device identification?

For example, each unprovisioned device contains a unique Device UUID, which can be used for device identification.

1.13 How is the unicast address assigned when the node provisioned by Provisioner contains multiple elements?

- Provisioner will assign an unicast address for the primary element of the node, and unicast address of the remaining elements are incremented one by one.
- For example: If an unprovisioned device has three elements, i.e. the primary element, the second element and the third element. After provisioning, the primary element address of the node is 0x0002 while the second element address is 0x0003, and the third element address is 0x0004.

1.14 How can Provisioner get and parse the Composition Data of nodes through Configuration Client Model?

- Provisioner can get the Composition Data of nodes using the *Configuration Client Model* API `esp_ble_mesh_config_client_set_state()` with `comp_data_get` in the parameter `esp_ble_mesh_cfg_client_get_state_t` set properly.
- Users can refer to the following code to parse the Composition Data:

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>

//test date: 0C001A0001000800030000010501000000800100001003103F002A00
//0C00 1A00 0100 0800 0300 0001 05 01 0000 0080 0100 0010 0310 3F002A00

// CID is 0x000C
// PID is 0x001A
// VID is 0x0001
// CRPL is 0x0008
// Features is 0x0003 - Relay and Friend features.
// Loc is "front" - 0x0100
// NumS is 5
// NumV is 1
// The Bluetooth SIG Models supported are: 0x0000, 0x8000, 0x0001, 0x1000,
→ 0x1003
// The Vendor Models supported are: Company Identifier 0x003F and Model_
→Identifier 0x002A

typedef struct {
    int16_t cid;
    int16_t pid;
    int16_t vid;
    int16_t crpl;
```

(continues on next page)

(continued from previous page)

```

    uint16_t features;
    uint16_t all_models;
    uint8_t sig_models;
    uint8_t vnd_models;
} esp_ble_mesh_composition_head;

typedef struct {
    uint16_t model_id;
    uint16_t vendor_id;
} tsModel;

typedef struct {
    // reserve space for up to 20 SIG models
    uint16_t SIG_models[20];
    uint8_t numSIGModels;

    // reserve space for up to 4 vendor models
    tsModel Vendor_models[4];
    uint8_t numVendorModels;
} esp_ble_mesh_composition_decode;

int decode_comp_data(esp_ble_mesh_composition_head *head, esp_ble_mesh_
↳composition_decode *data, uint8_t *mystr, int size)
{
    int pos_sig_base;
    int pos_vnd_base;
    int i;

    memcpy(head, mystr, sizeof(*head));

    if(size < sizeof(*head) + head->sig_models * 2 + head->vnd_models * 2
↳4) {
        return -1;
    }

    pos_sig_base = sizeof(*head) - 1;

    for(i = 1; i < head->sig_models * 2; i = i + 2) {
        data->SIG_models[i/2] = mystr[i + pos_sig_base] | (mystr[i + pos_
↳sig_base + 1] << 8);
        printf("%d: %4.4x\n", i/2, data->SIG_models[i/2]);
    }

    pos_vnd_base = head->sig_models * 2 + pos_sig_base;

    for(i = 1; i < head->vnd_models * 2; i = i + 2) {
        data->Vendor_models[i/2].model_id = mystr[i + pos_vnd_base] |
↳(mystr[i + pos_vnd_base + 1] << 8);
        printf("%d: %4.4x\n", i/2, data->Vendor_models[i/2].model_id);

        data->Vendor_models[i/2].vendor_id = mystr[i + pos_vnd_base + 2]
↳(mystr[i + pos_vnd_base + 3] << 8);
        printf("%d: %4.4x\n", i/2, data->Vendor_models[i/2].vendor_id);
    }

    return 0;
}

void app_main(void)
{
    esp_ble_mesh_composition_head head = {0};

```

(continues on next page)

(continued from previous page)

```

esp_ble_mesh_composition_decode data = {0};
uint8_t mystr[] = { 0x0C, 0x00, 0x1A, 0x00,
                   0x01, 0x00, 0x08, 0x00,
                   0x03, 0x00, 0x00, 0x01,
                   0x05, 0x01, 0x00, 0x00,
                   0x00, 0x80, 0x01, 0x00,
                   0x00, 0x10, 0x03, 0x10,
                   0x3F, 0x00, 0x2A, 0x00};

int ret;

ret = decode_comp_data(&head, &data, mystr, sizeof(mystr));
if (ret == -1) {
    printf("decode_comp_data error");
}
}

```

1.15 How can Provisioner further configure nodes through obtained Composition Data?

Provisioner do the following configuration by calling the *Configuration Client Model* API `esp_ble_mesh_config_client_set_state()`.

- Add AppKey to nodes with `app_key_add` in the parameter `esp_ble_mesh_cfg_client_set_state_t` set properly.
- Add subscription address to the models of nodes with `model_sub_add` in the parameter `esp_ble_mesh_cfg_client_set_state_t` set properly.
- Set publication information to the models of nodes with `model_pub_set` in the parameter `esp_ble_mesh_cfg_client_set_state_t` set properly.

1.16 Can nodes add corresponding configurations for themselves?

This method can be used in special cases like testing period.

- Here is an example to show nodes add new group addresses for their models.

```

esp_err_t example_add_fast_prov_group_address(uint16_t model_id, uint16_t_
→group_addr)
{
    const esp_ble_mesh_comp_t *comp = NULL;
    esp_ble_mesh_elem_t *element = NULL;
    esp_ble_mesh_model_t *model = NULL;
    int i, j;

    if (!ESP_BLE_MESH_ADDR_IS_GROUP(group_addr)) {
        return ESP_ERR_INVALID_ARG;
    }

    comp = esp_ble_mesh_get_composition_data();
    if (!comp) {
        return ESP_FAIL;
    }

    for (i = 0; i < comp->element_count; i++) {
        element = &comp->elements[i];
        model = esp_ble_mesh_find_sig_model(element, model_id);
        if (!model) {
            continue;
        }
        for (j = 0; j < ARRAY_SIZE(model->groups); j++) {
            if (model->groups[j] == group_addr) {
                break;
            }
        }
    }
}

```

(continues on next page)

(continued from previous page)

```

    }
}
if (j != ARRAY_SIZE(model->groups)) {
    ESP_LOGW(TAG, "%s: Group address already exists, element_
↪index: %d", __func__, i);
    continue;
}
for (j = 0; j < ARRAY_SIZE(model->groups); j++) {
    if (model->groups[j] == ESP_BLE_MESH_ADDR_UNASSIGNED) {
        model->groups[j] = group_addr;
        break;
    }
}
if (j == ARRAY_SIZE(model->groups)) {
    ESP_LOGE(TAG, "%s: Model is full of group addresses, element_
↪index: %d", __func__, i);
}
}

return ESP_OK;
}

```

Note: When the NVS storage of the node is enabled, group address added and AppKey bound by this method will not be saved in the NVS when the device is powered off currently. These configuration information can only be saved if they are configured by Configuration Client Model.

1.17 How does Provisioner control nodes by grouping?

Generally there are two approaches to implement group control in ESP-BLE-MESH network, group address approach and virtual address approach. And supposing there are 10 devices, i.e., five devices with blue lights and five devices with red lights.

- Method 1: 5 blue lights can subscribe to a group address, 5 red lights subscribe to another one. By sending messages to different group addresses, Provisioner can realize group control.
- Method 2: 5 blue lights can subscribe to a virtual address, 5 red lights subscribe to another one. By sending messages to different virtual addresses, Provisioner can realize group control.

1.18 How does Provisioner add nodes to multiple subnets?

Provisioner can add multiple NetKeys to nodes during configuration, and nodes sharing the same NetKey belong to the same subnet. Provisioner can communicate with nodes on different subnets by using different NetKeys.

1.19 How does Provisioner know if a node in the mesh network is offline?

Node offline is usually defined as: the condition that the node cannot be properly communicated with other nodes in the mesh network due to power failure or some other reasons.

There is no connection between nodes and nodes in the ESP-BLE-MESH network. They communicate with each other through advertising channels.

An example is given here to show how to detect a node is offline by Provisioner.

- The node can periodically send heartbeat messages to Provisioner. And if Provisioner failed to receive heartbeat messages in a certain period, the node is considered to be offline.

Note: The heartbeat message should be designed into a single package (less than 11 bytes), so the transmission and reception of it can be more efficient.

1.20 What operations should be performed when Provisioner removes nodes from the network?

Usually when Provisioner tries to remove node from the mesh network, the procedure includes three main steps:

- Firstly, Provisioner adds the node that need to be removed to the “blacklist” .
- Secondly, Provisioner performs the *Key Refresh procedure*.
- Lastly, the node performs node reset procedure, and switches itself to an unprovisioned device.

1.21 In the Key Refresh procedure, how does Provisioner update the Netkey owned by nodes?

- Provisioner updates the NetKey of nodes using the *Configuration Client Model* API `esp_ble_mesh_config_client_set_state()` with `net_key_update` in the parameter `esp_ble_mesh_cfg_client_set_state_t` set properly.
- Provisioner updates the AppKey of nodes using the *Configuration Client Model* API `esp_ble_mesh_config_client_set_state()` with `app_key_update` in the parameter `esp_ble_mesh_cfg_client_set_state_t` set properly.

1.22 How does Provisioner manage nodes in the mesh network?

ESP-BLE-MESH implements several functions related to basic node management in the example, such as `esp_ble_mesh_store_node_info()`. And ESP-BLE-MESH also provides the API `esp_ble_mesh_provisioner_set_node_name()` which can be used to set the node’s local name and the API `esp_ble_mesh_provisioner_get_node_name()` which can be used to get the node’s local name.

1.23 What does Provisioner need when trying to control the server model of nodes?

Provisioner must include corresponding client model before controlling the server model of nodes.

Provisioner shall add its local NetKey and AppKey.

- Provisioner add NetKey by calling the API `esp_ble_mesh_provisioner_add_local_net_key()`.
- Provisioner add AppKey by calling the API `esp_ble_mesh_provisioner_add_local_app_key()`.

Provisioner shall configure its own client model.

- Provisioner bind AppKey to its own client model by calling the API `esp_ble_mesh_provisioner_bind_app_key_to_local_model()`.

1.24 How does Provisioner control the server model of nodes?

ESP-BLE-MESH supports all SIG-defined client models. Provisioner can use these client models to control the server models of nodes. And the client models are divided into 6 categories with each category has the corresponding functions.

- Configuration Client Model
 - The API `esp_ble_mesh_config_client_get_state()` can be used to get the `esp_ble_mesh_cfg_client_get_state_t` values of Configuration Server Model.
 - The API `esp_ble_mesh_config_client_set_state()` can be used to set the `esp_ble_mesh_cfg_client_set_state_t` values of Configuration Server Model.
- Health Client Model
 - The API `esp_ble_mesh_health_client_get_state()` can be used to get the `esp_ble_mesh_health_client_get_state_t` values of Health Server Model.

- The API `esp_ble_mesh_health_client_set_state()` can be used to set the `esp_ble_mesh_health_client_set_state_t` values of Health Server Model.
- **Generic Client Models**
 - The API `esp_ble_mesh_generic_client_get_state()` can be used to get the `esp_ble_mesh_generic_client_get_state_t` values of Generic Server Models.
 - The API `esp_ble_mesh_generic_client_set_state()` can be used to set the `esp_ble_mesh_generic_client_set_state_t` values of Generic Server Models.
- **Lighting Client Models**
 - The API `esp_ble_mesh_light_client_get_state()` can be used to get the `esp_ble_mesh_light_client_get_state_t` values of Lighting Server Models.
 - The API `esp_ble_mesh_light_client_set_state()` can be used to set the `esp_ble_mesh_light_client_set_state_t` values of Lighting Server Models.
- **Sensor Client Models**
 - The API `esp_ble_mesh_sensor_client_get_state()` can be used to get the `esp_ble_mesh_sensor_client_get_state_t` values of Sensor Server Model.
 - The API `esp_ble_mesh_sensor_client_set_state()` can be used to set the `esp_ble_mesh_sensor_client_set_state_t` values of Sensor Server Model.
- **Time and Scenes Client Models**
 - The API `esp_ble_mesh_time_scene_client_get_state()` can be used to get the `esp_ble_mesh_time_scene_client_get_state_t` values of Time and Scenes Server Models.
 - The API `esp_ble_mesh_time_scene_client_set_state()` can be used to set the `esp_ble_mesh_time_scene_client_set_state_t` values of Time and Scenes Server Models.

2. Node Development

2.1 What kind of models are included by nodes?

- In ESP-BLE-MESH, nodes are all composed of a series of models with each model implements some functions of the node.
- Model has two types, client model and server model. Client model can get and set the states of server model.
- Model can also be divided into SIG model and vendor model. All behaviors of SIG models are officially defined while behaviors of vendor models are defined by users.

2.2 Is the format of messages corresponding to each model fixed?

- Messages, which consist of opcode and payload, are divided by opcode.
- The type and the format of the messages corresponding to models are both fixed, which means the messages transmitted between models are fixed.

2.3 Which functions can be used to send messages with the models of nodes?

- For client models, users can use the API `esp_ble_mesh_client_model_send_msg()` to send messages.
- For server models, users can use the API `esp_ble_mesh_server_model_send_msg()` to send messages.
- For publication, users call the API `esp_ble_mesh_model_publish()` to publish messages.

2.4 How to achieve the transmission of messages without packet loss?

Acknowledged message is needed if users want to transmit messages without packet loss. The default time to wait for corresponding response is set in `CONFIG_BLE_MESH_CLIENT_MSG_TIMEOUT`. If the sender waits for the response until the timer expires, the corresponding timeout event would be triggered.

Note: Response timeout can be set in the API `esp_ble_mesh_client_model_send_msg()`. The default value (4 seconds) would be applied if the parameter `msg_timeout` is set to **0**.

2.5 How to send unacknowledged messages?

For client models, users can use the API `esp_ble_mesh_client_model_send_msg()` with the parameter `need_rsp` set to `false` to send unacknowledged messages.

For server models, the messages sent by using the API `esp_ble_mesh_server_model_send_msg()` are always unacknowledged messages.

2.6 How to add subscription address to models?

Subscription address can be added through Configuration Client Model.

2.7 What is the difference between messages sent and published by models?

Messages sent by calling the API `esp_ble_mesh_client_model_send_msg()` or `esp_ble_mesh_server_model_send_msg()` will be sent in the duration determined by the Network Transmit state.

Messages published by calling the API `esp_ble_mesh_model_publish()` will be published determined by the Model Publication state. And the publication of messages is generally periodic or with a fixed number of counts. The publication period and publication count are controlled by the Model Publication state, and can be configured through Configuration Client Model.

2.8 How many bytes can be carried when sending unsegmented messages?

The total payload length (which can be set by users) of unsegmented message is 11 octets, so if the opcode of the message is 2 octets, then the message can carry 9-octets of valid information. For vendor messages, due to the 3-octets opcode, the remaining payload length is 8 octets.

2.9 When should the Relay feature of nodes be enabled?

Users can enable the Relay feature of all nodes when nodes detected in the mesh network are sparse.

For dense mesh network, users can choose to just enable the Relay feature of several nodes.

And users can enable the Relay feature by default if the mesh network size is unknown.

2.10 When should the Proxy feature of node be enabled?

If the unprovisioned device is expected to be provisioned by a phone, then it should enable the Proxy feature since almost all the phones do not support sending ESP-BLE-MESH packets through advertising bearer currently. And after the unprovisioned device is provisioned successfully and becoming a Proxy node, it will communicate with the phone using GATT bearer and using advertising bearer to communicate with other nodes in the mesh network.

2.11 How to use the Proxy filter?

The Proxy filter is used to reduce the number of Network PDUs exchanged between a Proxy Client (e.g. the phone) and a Proxy Server (e.g. the node). And with the Proxy filter, Proxy Client can explicitly request to receive only mesh messages with certain destination addresses from Proxy Server.

2.12 When a message can be relayed by a Relay node?

If a message need to be relayed, the following conditions should be met.

- The message is in the mesh network.
- The message is not sent to the unicast address of the node.
- The value of TTL in the message is greater than 1.

2.13 If a message is segmented into several segments, should the other Relay nodes just relay when one of these segments is received or wait until the message is received completely?

Relay nodes will forward segments when one of them are received rather than keeping waiting until all the segments are received.

2.14 What is the principle of reducing power consumption using Low Power feature?

- When the radio is turned on for listening, the device is consuming energy. When low power feature of the node is enabled, it will turn off its radio in the most of the time.
- And cooperation is needed between low power node and friend node, thus low power node can receive messages at an appropriate or lower frequency without the need to keep listening.
- When there are some new messages for low power node, its friend node will store the messages for it. And low power node can poll friend nodes to see if there are new messages at a fixed interval.

2.15 How to continue the communication on the network after powering-down and powering-up again?

Enable the configuration `Store ESP-BLE-MESH Node configuration persistently` in `menuconfig`.

2.16 How to send out the self-test results of nodes?

It is recommended that nodes can publish its self-test results periodically through Health Server Model.

2.17 How to transmit information between nodes?

One possible application scenario for transmitting information between nodes is that spray nodes would be triggered once smoke alarm detected high smoke concentration. There are two approaches in implementation.

- Approach 1 is that spray node subscribes to a group address. When smoke alarm detects high smoke concentration, it will publish a message whose destination address is the group address which has been subscribed by spray node.
- Approach 2 is that Provisioner can configure the unicast address of spray node to the smoke alarm. When high smoke concentration is detected, smoke alarm can use send messages to the spray node with the spray node's unicast address as the destination address.

2.18 Is gateway a must for nodes communication?

- Situation 1: nodes only communicate within the mesh network. In this situation, no gateway is need. ESP-BLE-MESH network is a flooded network, messages in the network have no fixed paths, and nodes can communicate with each other freely.
- Situation 2: if users want to control the nodes remotely, for example turn on some nodes before getting home, then a gateway is needed.

2.19 When will the IV Update procedure be performed?

IV Update procedure would be performed once sequence number of messages sent detected by the bottom layer of node reached a critical value.

2.20 How to perform IV Update procedure?

Nodes can perform IV Update procedure with Secure Network Beacon.

3. ESP-BLE-MESH and Wi-Fi Coexistence

3.1 Which modes does Wi-Fi support when it coexists with ESP-BLE-MESH?

Currently only Wi-Fi station mode supports the coexistence.

3.2 Why is the Wi-Fi throughput so low when Wi-Fi and ESP-BLE-MESH coexist?

The `ESP32-DevKitC` board without PSRAM can run properly but the throughput of it is low since it has no PSRAM. When Bluetooth and Wi-Fi coexist, the throughput of ESP32-DevKitC with PSRAM can be stabilized to more than 1Mbps.

And some configurations in `menuconfig` shall be enabled to support PSRAM.

- `ESP32-specific` --> Support for external, SPI-connected RAM --> Try to allocate memories of Wi-Fi and LWIP...
- `Bluetooth` --> `Bluedroid Enable` --> BT/BLE will first malloc the memory from the PSRAM
- `Bluetooth` --> `Bluedroid Enable` --> Use dynamic memory allocation in BT/BLE stack.
- `Bluetooth` --> `Bluetooth controller` --> BLE full scan feature supported.
- `Wi-Fi` --> Software controls Wi-Fi/Bluetooth coexistence --> Wi-Fi

4. Fast Provisioning

4.1 Why is fast provisioning needed?

Normally when they are several unprovisioned devices, users can provision them one by one. But when it comes to a large number of unprovisioned devices (e.g. 100), provisioning them one by one will take huge amount of time. With fast provisioning, users can provision 100 unprovisioned devices in about 50 seconds.

4.2 Why EspBleMesh App would wait for a long time during fast provisioning?

After the App provisioned one Proxy node, it will disconnect from the App during fast provisioning, and reconnect with the App when all the nodes are provisioned.

4.3 Why is the number of node addresses displayed in the App is more than that of existing node addresses?

Each time after a fast provisioning process, and before starting a new one, the node addresses in the App should be cleared, otherwise the number of the node address will be incorrect.

4.4 What is the usage of the count value which was input in EspBleMesh App?

The **count** value is provided to the Proxy node which is provisioned by the App so as to determine when to start Proxy advertising in advance.

4.5 When will Configuration Client Model of the node running fast_prov_server example start to work?

Configuration Client Model will start to work after the Temporary Provisioner functionality is enabled.

4.6 Will the Temporary Provisioner functionality be enabled all the time?

After the nodes receive messages used to turn on/off lights, all the nodes will disable its Temporary Provisioner functionality and become nodes.

5. Log Help

You can find meaning of errors or warnings when they appear at the bottom of ESP-BLE-MESH stack.

5.1 What is the meaning of warning **ran out of retransmit attempts?**

When the node transmits a segmented message, and due to some reasons, the receiver doesn't receive the complete message. Then the node will retransmit the message. When the retransmission count reaches the maximum number, which is 4 currently, then this warning will appear.

5.2 What is the meaning of warning **Duplicate found in Network Message Cache?**

When the node receives a message, it will compare the message with the ones stored in the network cache. If the same has been found in the cache, which means it has been received before, then the message will be dropped.

5.3 What is the meaning of warning **Incomplete timer expired?**

When the node doesn't receive all the segments of a segmented message during a certain period (e.g. 10 seconds), then the Incomplete timer will expire and this warning will appear.

5.4 What is the meaning of warning **No matching TX context for ack?**

When the node receives a segment ack and it doesn't find any self-send segmented message related with this ack, then this warning will appear.

5.5 What is the meaning of warning **No free slots for new incoming segmented messages?**

When the node has no space for receiving new segmented message, this warning will appear. Users can make the space larger through the configuration `CONFIG_BLE_MESH_RX_SEG_MSG_COUNT`.

5.6 What is the meaning of error **Model not bound to Appkey 0x0000 ?**

When the node sends messages with a model and the model has not been bound to the AppKey with AppKey Index 0x000, then this error will appear.

5.7 What is the meaning of error **Busy sending message to DST xxxx?**

This error means client model of the node has transmitted a message to the target node and now is waiting for a response, users can not send messages to the same node with the same unicast address. After the corresponding response is received or timer is expired, then another message can be sent.

6. Example Help

6.1 How are the ESP-BLE-MESH callback functions classified?

- The API `esp_ble_mesh_register_prov_callback()` is used to register callback function used to handle provisioning and networking related events.
- The API `esp_ble_mesh_register_config_client_callback()` is used to register callback function used to handle Configuration Client Model related events.
- The API `esp_ble_mesh_register_config_server_callback()` is used to register callback function used to handle Configuration Server Model related events.

- The API `esp_ble_mesh_register_health_client_callback()` is used to register callback function used to handle Health Client Model related events.
- The API `esp_ble_mesh_register_health_server_callback()` is used to register callback function used to handle Health Server Model related events.
- The API `esp_ble_mesh_register_generic_client_callback()` is used to register callback function used to handle Generic Client Models related events.
- The API `esp_ble_mesh_register_light_client_callback()` is used to register callback function used to handle Lighting Client Models related events.
- The API `esp_ble_mesh_register_sensor_client_callback()` is used to register callback function used to handle Sensor Client Model related events.
- The API `esp_ble_mesh_register_time_scene_client_callback()` is used to register callback function used to handle Time and Scenes Client Models related events.
- The API `esp_ble_mesh_register_custom_model_callback()` is used to register callback function used to handle vendor model and unrealized server models related events.

7. Others

7.1 How to print the message context?

The examples use `ESP_LOG_BUFFER_HEX()` to print the message context while the ESP-BLE-MESH protocol stack uses `bt_hex()`.

7.2 Which API can be used to restart ESP32?

The API `esp_restart()`.

7.3 How to monitor the remaining space of the stack of a task?

The API `vTaskList()` can be used to print the remaining space of the task stack periodically.

7.4 How to change the level of log without changing the menuconfig output level?

The API `esp_log_level_set()` can be used to change the log output level rather than using menuconfig to change it.

ESP-BLE-MESH Terminology

Table 12: Table 1 ESP-BLE-MESH Terminology - Role

| Term | Official Definition | Detailed Explanation |
|----------------------|--|--|
| Unprovisioned Device | A device that is not a member of a mesh network is known as an unprovisioned device. | Examples: lighting devices, temperature control devices, manufacturing equipments and electric doors, etc. |
| Node | A node is a provisioned device. | The role of unprovisioned device will change to node after being provisioned to ESP-BLE-MESH network. Nodes (such as lighting devices, temperature control devices, manufacturing equipments, and electric doors) are devices that can send, receive, or relay messages in ESP-BLE-MESH network, and they can optionally support one or more subnets. |
| Relay Node | A node that supports the Relay feature and has the Relay feature enabled is known as a Relay node. | Relay nodes can receive and resend ESP-BLE-MESH messages, so the messages can be transferred further. Users can decide whether or not to enable forwarding function of nodes according to nodes' status. Messages can be relayed for multiple times, and each relay is considered as a "hop". Messages can hop up to 126 times, which is enough for message transmission in a wide area. |
| Proxy Node | A node that supports the Proxy feature and has the Proxy feature enabled is known as a Proxy node. | Proxy nodes receive messages from one bearer (it generally includes advertising bearer and GATT bearer) and resend it from another one. The purpose is to connect communication equipments that only support GATT bearer to ESP-BLE-MESH network. Generally, mobile apps need a Proxy node to access Mesh network. Without Proxy nodes, mobile apps cannot communicate with members in Mesh network. |
| Friend Node | A node that supports the Friend feature, has the Friend feature enabled, and has a friendship with a node that supports the Low Power feature is known as a Friend node. | Friend node, like the backup of Low Power node (LPN), can store messages that are sent to Low Power node and security updates; the stored information will be transferred to Low Power node when Low Power node needs it. Low Power node must establish "friendship" with another node that supports the Friend Feature to reduce duty cycle of its receiver, thus power consumption of Low Power node can be reduced. Low Power node needs to find a Friend node to establish a friendship with it. The process involved is called "friendship establishment". Cooperation between Low Power node and Friend nodes enables Low Power node to schedule the use of the radio, thus Low Power node can receive messages at an appropriate or lower frequency without the need of keeping listening. Low Power node will poll Friend node to see if there is new message. |
| Low Power Node | A node that supports the Low Power feature and has a friendship with a node that supports the Friend feature is known as a Low Power node. | By polling, Low Power node gets information from Friend node, such as messages, security updates, and etc. |
| Provisioner | A node that is capable of adding a device to a mesh network. | The device that can provision unprovisioned devices is called a Provisioner. This process usually needs to be implemented through an app that is typically provided by the product manufacturer and can be used on a gateway, a smartphone, tablet or other carriers. |

Table 13: Table 2 ESP-BLE-MESH Terminology - Composition

| Term | Official Definition | Detailed Explanation |
|------------------------|---|--|
| State | A value representing a condition of an element that is exposed by an element of a node. | Each node in a ESP-BLE-MESH network has an independent set of state values that indicate certain states of the device, like brightness, and color of lighting device. Change of state value will lead to change of the physical state of devices. For example, changing the on/off state of a device is actually turning on/off the device. |
| Model | A model defines the basic functionality of a node. | A node may contain multiple models, and each model defines basic functionalities of nodes, like the states needed by the nodes, the messages controlling the states, and actions resulted from messages handling. The function implementation of the nodes is based on models, which can be divided into SIG Model and Vendor Model, with the former defined by SIG and latter defined by users. |
| Element | An addressable entity within a device. | A node can contain one or more elements, with each having a unicast address and one or more models, and the models contained by the same element must not be the same. |
| Composition Data State | The Composition Data state contains information about a node, the elements it includes, and the supported models. | By reading the value of the Composition Data state, users can know basic information of the node, such as the number of elements, and the models in each element. Provisioner gets this message to further provision the device, such as configuring subscription address and publishing address of nodes. |

Table 14: Table 3 ESP-BLE-MESH Terminology - Features

| Term | Official Definition | Detailed Explanation |
|-------------------|---|--|
| Low Power Feature | The ability to operate within a mesh network at significantly reduced receiver duty cycles only in conjunction with a node supporting the Friend feature. | Low Power feature reduces power consumption of nodes. When a Low Power node is searching for a Friend node, and there are multiple Friend nodes nearby, it selects the most suitable Friend node through algorithm. |
| Friend Feature | The ability to help a node supporting the Low Power feature to operate by storing messages destined for those nodes. | By enabling friend feature, the node can help to store information for Low Power node. The nodes enabled with friend feature may cause more power and memory consumption. |
| Relay Feature | The ability to receive and retransmit mesh messages over the advertising bearer to enable larger networks. | The relay feature enables ESP-BLE-MESH messages to hop among nodes for multiple times, and the transmission distance can exceed the range of direct radio transmission between two nodes, thereby covering the entire network. When a node is enabled with the relay feature to relay messages, it only relays the messages of its own subnet, and does not relay the messages of other subnets. The data integrity will not be considered when the node enabled with relay feature relays segmented messages. The node would relay every segmented message once it receives one rather than waiting for the complete message. |
| Proxy Feature | The ability to receive and retransmit mesh messages between GATT and advertising bearers. | The purpose of the proxy feature is to allow nodes without an advertising bearer to access the ESP-BLE-MESH network. The proxy feature is typically used in nodes that need to connect to mobile apps. |

Table 15: Table 4 ESP-BLE-MESH Terminology - Provisioning

| Term | Official Definition | Detailed Explanation |
|-----------------------|---|--|
| PB-ADV | PB-ADV is a provisioning bearer used to provision a device using Generic Provisioning PDUs over the advertising channels. | PB-ADV transfers packets generated during the provisioning process over the advertising channels. This way can only be used for provisioning when provisioner and unprovisioned device both support PB-ADV. |
| PB-GATT | PB-GATT is a provisioning bearer used to provision a device using Proxy PDUs to encapsulate Provisioning PDUs within the Mesh Provisioning Service. | PB-GATT uses connection channels to transfer packets generated during the provisioning process. If an unprovisioned device wants to be provisioned through this method, it needs to implement the related Mesh Provisioning Service. Unprovisioned devices which don't implement such service cannot be provisioned into mesh network through PB-GATT bearer. |
| Provisioning | Provisioning is a process of adding an unprovisioned device to a mesh network, managed by a Provisioner. | The process of provisioning turns the “unprovisioned device” into a “node”, making it a member of the ESP-BLE-MESH network. |
| Authentication Method | Authentication is a step during the provisioning of nodes. | There are four authentication methods for unprovisioned devices: Output OOB, Input OOB, Static OOB, and No OOB. |
| Input OOB | Input Out-of-Band | For example, a Provisioner generates and displays a random number, and then prompts users to take appropriate actions to input the random number into the unprovisioned device. Taking lighting switch as an example, users can press the button for several times in a certain period of time to input the random number displayed on the Provisioner. Authentication method of the Input OOB is similar to that of Output OOB, but the role of the device is reversed. |
| Output OOB | Output Out-of-Band | For example, an unprovisioned device will choose a random number and output the number in a way that is compatible with its functionality. If the unprovisioned device is a bulb, it can flash a specified number of times. If the unprovisioned device has an LCD screen, the random number can display as a multi-digit value. Users who start provisioning should input the observed number to authenticate the unprovisioned device. |
| Static OOB | Static Out-of-Band | Authentication method of Static OOB: use Static OOB information. Use 0 as Static OOB information if No OOB information is needed. Use Static OOB information to authenticate devices which are going through provisioning if OOB information is needed. |
| No OOB | No Out-of-Band | Authentication method of No OOB: Set the value of the Static OOB field to 0. Using this way is like not authenticating the unprovisioned devices. |

Table 16: Table 5 ESP-BLE-MESH Terminology - Address

| Term | Official Definition | Detailed Explanation |
|--------------------|---|---|
| Unassigned Address | This is a special address type, with a value of 0x0000. Its use indicates that an Element has not yet been configured or had a Unicast Address assigned to it. | The addresses owned by elements which has not been configured yet or no address has been allocated are unassigned addresses. These elements will not be used for messages transfer because they have no fixed address. Unassigned address is recommended to set as the value of the address before setting the address of user code. |
| Unicast Address | A unicast address is a unique address allocated to each element. | During provisioning, the Provisioner will assign a unicast address to each element of node within the life cycle of the nodes in the network. A unicast address may appear in the source/destination address field of a message. Messages sent to a unicast address can only be processed by the element that owns the unicast address. |
| Virtual Address | A virtual address represents a set of destination addresses. Each virtual address logically represents a Label UUID, which is a 128-bit value that does not have to be managed centrally. | Associated with specific UUID labels, a virtual address may serve as the publishing or subscription address of the model. A UUID label is a 128-bit value associated with elements of one or more nodes. For virtual addresses, the 15th and 14th bits are set to 1 and 0 respectively; bits from 13th to 0 are set to hash values (providing 16384 hash values). The hash is a derivation of the Label UUID. To use subscribing elements to check the full 128-bit UUID is very inefficient while hash values provide a more efficient way to determine which elements that which messages are finally sent to. |
| Group Address | A group address is an address that is programmed into zero or more elements | Group address is another kind of multicast address in the ESP-BLE-MESH network, which is usually used to group nodes. A message sent to the all-proxies address shall be processed by the primary element of all nodes that have the proxy functionality enabled. A message sent to the all-friends address shall be processed by the primary element of all nodes that have the friend functionality enabled. A message sent to the all-relays address shall be processed by the primary element of all nodes that have the relay functionality enabled. A message sent to the all-nodes address shall be processed by the primary element of all nodes. |

Table 17: Table 6 ESP-BLE-MESH Terminology - Security

| Term | Official Definition | Detailed Explanation |
|---------------------------|---|---|
| Device Key (DevKey) | There is also a device key, which is a special application key that is unique to each node, is known only to the node and a Configuration Client, and is used to secure communications between the node and a Configuration Client. | The device key enables you to provision the devices, configure the nodes. The device key is used to encrypt Configuration Messages, i.e. the message transferred between the Provisioner and the node when the device is configured. |
| Application Key (App-Key) | Application keys are used to secure communications at the upper transport layer. | Application key is used for decryption of application data before delivering application data to application layer and encryption of them during the delivery of application layer. Some nodes in the network have a specific purpose and can restrict access to potentially sensitive data based on the needs of the application. With specific application keys, these nodes are associated with specific applications. Generally speaking, the fields using different application keys include security (access control of buildings, machine rooms and CEO offices), lighting (plant, exterior building and sidewalks) and HVAC systems. Application keys are bound to Network keys. This means application keys are only used in a context of a Network key they are bound to. An application key shall only be bound to a single Network key. |
| Master Security Material | The master security material is derived from the network key (NetKey) and can be used by other nodes in the same network. Messages encrypted with master security material can be decoded by any node in the same network. | The corresponding friendship messages encrypted with the friendship security material: 1. Friend Poll, 2. Friend Update, 3. Friend Subscription List, add/delete/confirm, 4. The Stored Messages” sent by friend nodes to Low Power node. The corresponding friendship messages encrypted with the master security material: 1. Friend Clear, 2. Friend Clear Confirm. Based on the setup of the applications, the messages sent from the Low Power node to the friend nodes will be encrypted with the friendship security material or master security material, with the former being used by the messages transmitted between Low Power node and friend nodes and the latter being used by other network messages. |

Table 18: Table 7 ESP-BLE-MESH Terminology - Message

| Term | Official Definition | Detailed Explanation |
|-------------------------------|---|---|
| Reassembly / Segmentation | Segmentation and reassembly (SAR) is a method of communication network, which is divided into small units before transmitting packets and reassembled in a proper order at the communication receiving end. | The lower transport layer will automatically segment the message whose size is too big. The receiving end will return a response message, and the transmitting end will send the data packet again that the receiving end does not receive according to the response message. This is automatically completed by the lower transport layer. Unsegmented messages have at most 15 bytes, of which 4 bytes are transMIC, so the remaining is 11 bytes; in the case of segmentation, there are 12 valid bytes in the first several packets, and 8 in the last one. Special case: A shorter packet requires mandatory segmentation from lower transport layer, in which case the valid byte is 8 bytes. |
| Unacknowledged / Acknowledged | There are two types of messages: Unacknowledged or Acknowledged | Based on the whether or not the receiving end needs to send the response message, the messages sent are divided into two kinds. The sending end should set the maximum number of retransmission. |

Table 19: Table 8 ESP-BLE-MESH Terminology - Foundation Models

| Term | Official Definition | Detailed Explanation |
|----------------------------|---|---|
| Configuration Server Model | This model is used to represent a mesh network configuration of a device. | The node must contain the Configuration Server Model, which is responsible for maintaining configuration-related states. The states that Configuration Server Model maintains include: NetKey List, AppKey List, Model to AppKey List, Node Identity, Key Refresh Phase, Heartbeat Publish, Heartbeat Subscription, Network Transmit, Relay Retransmit etc. |
| Configuration Client Model | The model is used to represent an element that can control and monitor the configuration of a node. | The Configuration Client Model uses messages to control the state maintained by the Configuration Server Model. The Provisioner must contain the Configuration Client Model, with which the configuration messages, like Configuration Composition Data Get can be sent. |
| Health Server Model | This model is used to represent a mesh network diagnostics of a device. | The Health Server Model is primarily used by devices to check their states and see if there is an error. The states maintained by Health Server model include: Current Fault, Registered Fault, Health Period, and Attention Timer. |
| Health Client Model | The model is used to represent an element that can control and monitor the health of a node. | The Health Client Model uses messages to control the state maintained by the Health Server Model. The model can get the self-test information of other nodes through the message "Health Fault Get" . |

Table 20: Table 9 ESP-BLE-MESH Terminology - Network Management

| Term | Official Definition | Detailed Explanation |
|---|---|---|
| Key Refresh procedure | This procedure is used when the security of one or more network keys and/or one or more of the application keys has been compromised or could be compromised. | Key Refresh Procedure is used to update network key and application key of ESP-BLE-MESH network. Key Refresh Procedure is used when the security of one or more network keys and/or one or more application keys is threatened or potentially threatened. Keys are usually updated after some nodes in the network are removed. |
| IV (Initialisation Vector) Update Procedure | A node can also use an IV Update procedure to signal to peer nodes that it is updating the IV Index. | The IV Update procedure is used to update the value of ESP-BLE-MESH network's IV Index. This value is related to the random number required for message encryption. To ensure that the value of the random number is not repeated, this value is periodically incremented. IV Index is a 32-bit value and a shared network resource. For example, all nodes in a mesh network share the same IV Index value. Starting from 0x00000000, the IV Index increments during the IV Update procedure and maintained by a specific process, ensuring the IV Index shared in the mesh network is the same. This can be done when the node believes that it has the risk of exhausting its sequence number, or when it determines that another node is nearly exhausting its sequence number. Note: The update time must not be less than 96 hours. It can be triggered when a secure network beacon is received, or when the node determines that its sequence number is greater than a certain value. |

For more terms, please see: [ESP-BLE-MESH Glossary of Terms](#).

Bluetooth SIG Documentation

- [BLE Mesh Core Specification](#)
- [BLE Mesh Model Specification](#)
- [An Intro to Bluetooth Mesh Part 1 / Part 2](#)

- [The Fundamental Concepts of Bluetooth Mesh Networking, Part 1 / Part 2](#)
- [Bluetooth Mesh Networking: Friendship](#)
- [Management of Devices in a Bluetooth Mesh Network](#)
- [Bluetooth Mesh Security Overview](#)
- [Provisioning a Bluetooth Mesh Network Part 1 / Part 2](#)

5.10 ESP-MESH

This guide provides information regarding the ESP-MESH protocol. Please see the [MESH API Reference](#) for more information about API usage.

5.10.1 Overview

ESP-MESH is a networking protocol built atop the Wi-Fi protocol. ESP-MESH allows numerous devices (henceforth referred to as nodes) spread over a large physical area (both indoors and outdoors) to be interconnected under a single WLAN (Wireless Local-Area Network). ESP-MESH is self-organizing and self-healing meaning the network can be built and maintained autonomously.

The ESP-MESH guide is split into the following sections:

1. [Introduction](#)
2. [ESP-MESH Concepts](#)
3. [Building a Network](#)
4. [Managing a Network](#)
5. [Data Transmission](#)
6. [Channel Switching](#)
7. [Performance](#)
8. [Further Notes](#)

5.10.2 Introduction

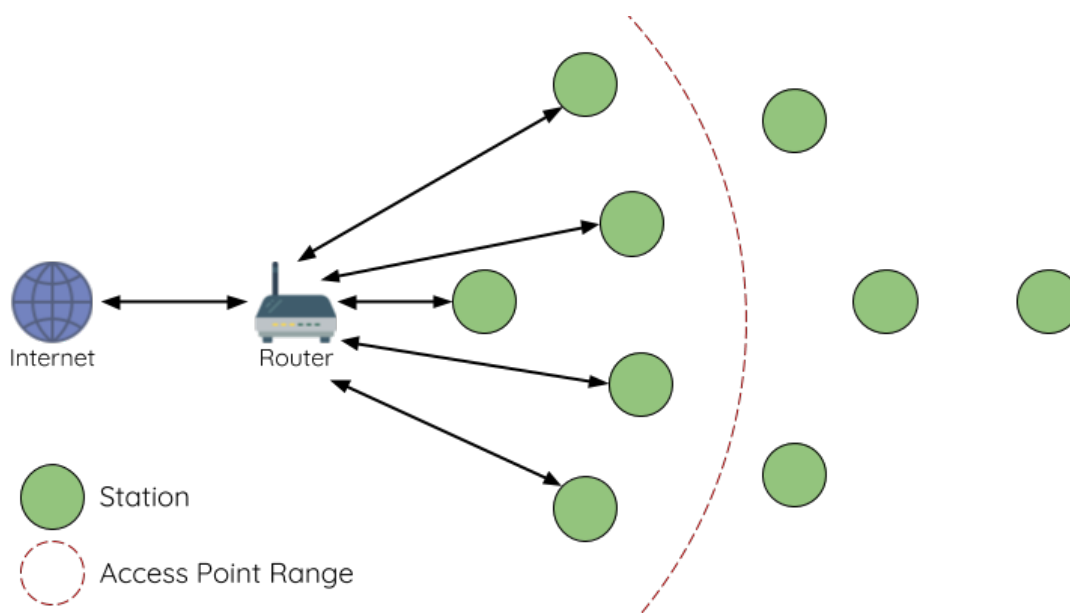


Fig. 16: Traditional Wi-Fi Network Architectures

A traditional infrastructure Wi-Fi network is a point-to-multipoint network where a single central node known as the access point (AP) is directly connected to all other nodes known as stations. The AP is responsible for arbitrating and

forwarding transmissions between the stations. Some APs also relay transmissions to/from an external IP network via a router. Traditional infrastructure Wi-Fi networks suffer the disadvantage of limited coverage area due to the requirement that every station must be in range to directly connect with the AP. Furthermore, traditional Wi-Fi networks are susceptible to overloading as the maximum number of stations permitted in the network is limited by the capacity of the AP.

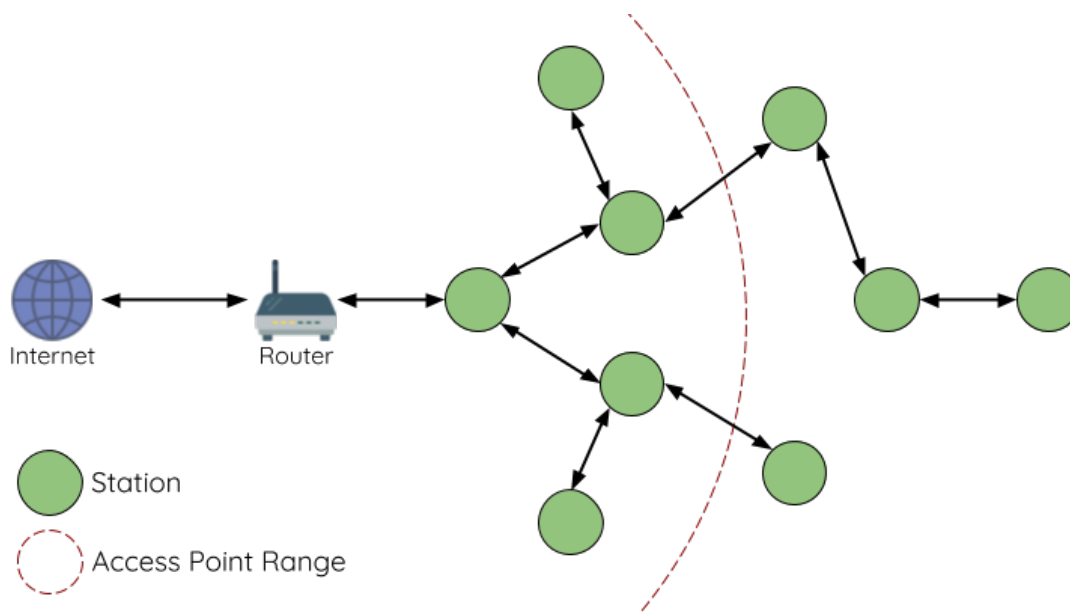


Fig. 17: ESP-MESH Network Architecture

ESP-MESH differs from traditional infrastructure Wi-Fi networks in that nodes are not required to connect to a central node. Instead, nodes are permitted to connect with neighboring nodes. Nodes are mutually responsible for relaying each others transmissions. This allows an ESP-MESH network to have much greater coverage area as nodes can still achieve interconnectivity without needing to be in range of the central node. Likewise, ESP-MESH is also less susceptible to overloading as the number of nodes permitted on the network is no longer limited by a single central node.

5.10.3 ESP-MESH Concepts

Terminology

| Term | Description |
|------------------------|--|
| Node | Any device that is or can be part of an ESP-MESH network |
| Root Node | The top node in the network |
| Child Node | A node X is a child node when it is connected to another node Y where the connection makes node X more distant from the root node than node Y (in terms of number of connections). |
| Parent Node | The converse notion of a child node |
| Sub-Child Node | Any node reachable by repeated proceeding from parent to child |
| Sibling Nodes | Nodes that share the same parent node |
| Connection | A traditional Wi-Fi association between an AP and a station. A node in ESP-MESH will use its station interface to associate with the softAP interface of another node, thus forming a connection. The connection process includes the authentication and association processes in Wi-Fi. |
| Up-stream Connection | The connection from a node to its parent node |
| Down-stream Connection | The connection from a node to one of its child nodes |
| Wireless Hop | The portion of the path between source and destination nodes that corresponds to a single wireless connection. A data packet that traverses a single connection is known as single-hop whereas traversing multiple connections is known as multi-hop . |
| Subnetwork | A subnetwork is subdivision of an ESP-MESH network which consists of a node and all of its descendant nodes. Therefore the subnetwork of the root node consists of all nodes in an ESP-MESH network. |
| MAC Address | Media Access Control Address used to uniquely identify each node or router within an ESP-MESH network. |
| DS | Distribution System (External IP Network) |

Tree Topology

ESP-MESH is built atop the infrastructure Wi-Fi protocol and can be thought of as a networking protocol that combines many individual Wi-Fi networks into a single WLAN. In Wi-Fi, stations are limited to a single connection with an AP (upstream connection) at any time, whilst an AP can be simultaneously connected to multiple stations (downstream connections). However ESP-MESH allows nodes to simultaneously act as a station and an AP. Therefore a node in ESP-MESH can have **multiple downstream connections using its softAP interface**, whilst simultaneously having a **single upstream connection using its station interface**. This naturally results in a tree network topology with a parent-child hierarchy consisting of multiple layers.

ESP-MESH is a multiple hop (multi-hop) network meaning nodes can transmit packets to other nodes in the network through one or more wireless hops. Therefore, nodes in ESP-MESH not only transmit their own packets, but simultaneously serve as relays for other nodes. Provided that a path exists between any two nodes on the physical layer (via one or more wireless hops), any pair of nodes within an ESP-MESH network can communicate.

Note: The size (total number of nodes) in an ESP-MESH network is dependent on the maximum number of layers permitted in the network, and the maximum number of downstream connections each node can have. Both of these variables can be configured to limit the size of the network.

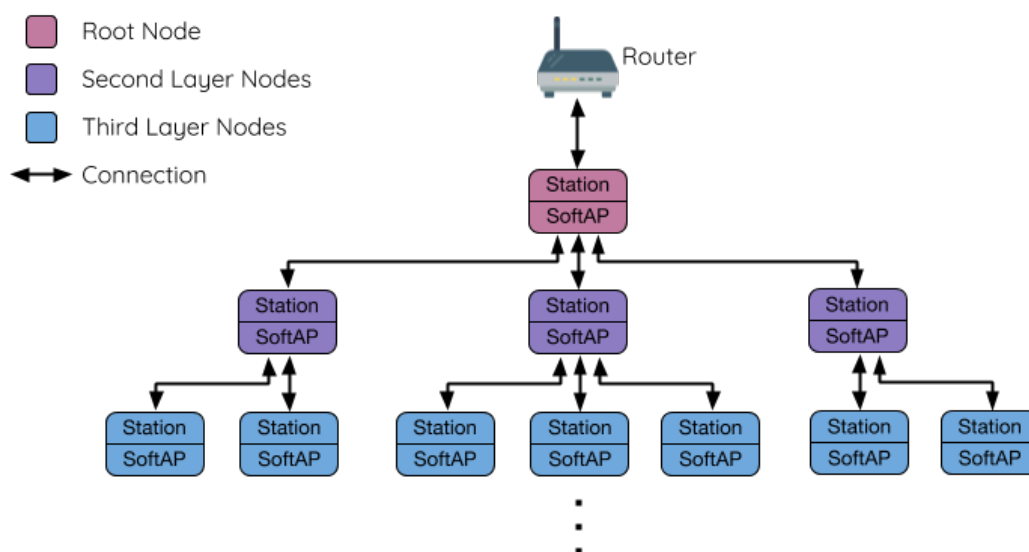


Fig. 18: ESP-MESH Tree Topology

Node Types

Root Node: The root node is the top node in the network and serves as the only interface between the ESP-MESH network and an external IP network. The root node is connected to a conventional Wi-Fi router and relays packets to/from the external IP network to nodes within the ESP-MESH network. **There can only be one root node within an ESP-MESH network** and the root node's upstream connection may only be with the router. Referring to the diagram above, node A is the root node of the network.

Leaf Nodes: A leaf node is a node that is not permitted to have any child nodes (no downstream connections). Therefore a leaf node can only transmit or receive its own packets, but cannot forward the packets of other nodes. If a node is situated on the network's maximum permitted layer, it will be assigned as a leaf node. This prevents the node from forming any downstream connections thus ensuring the network does not add an extra layer. Some nodes without a softAP interface (station only) will also be assigned as leaf nodes due to the requirement of a softAP interface for any downstream connections. Referring to the diagram above, nodes L/M/N are situated on the networks maximum permitted layer hence have been assigned as leaf nodes .

Intermediate Parent Nodes: Connected nodes that are neither the root node or a leaf node are intermediate parent nodes. An intermediate parent node must have a single upstream connection (a single parent node), but can have zero to multiple downstream connections (zero to multiple child nodes). Therefore an intermediate parent node can transmit and receive packets, but also forward packets sent from its upstream and downstream connections. Referring to the diagram above, nodes B to J are intermediate parent nodes. **Intermediate parent nodes without downstream connections such as nodes E/F/G/I/J are not equivalent to leaf nodes** as they are still permitted to form downstream connections in the future.

Idle Nodes: Nodes that have yet to join the network are assigned as idle nodes. Idle nodes will attempt to form an upstream connection with an intermediate parent node or attempt to become the root node under the correct circumstances (see [Automatic Root Node Selection](#)). Referring to the diagram above, nodes K and O are idle nodes.

Beacon Frames & RSSI Thresholding

Every node in ESP-MESH that is able to form downstream connections (i.e. has a softAP interface) will periodically transmit Wi-Fi beacon frames. A node uses beacon frames to allow other nodes to detect its presence and know of its status. Idle nodes will listen for beacon frames to generate a list of potential parent nodes, one of which the idle node will form an upstream connection with. ESP-MESH uses the Vendor Information Element to store metadata such as:

- Node Type (Root, Intermediate Parent, Leaf, Idle)

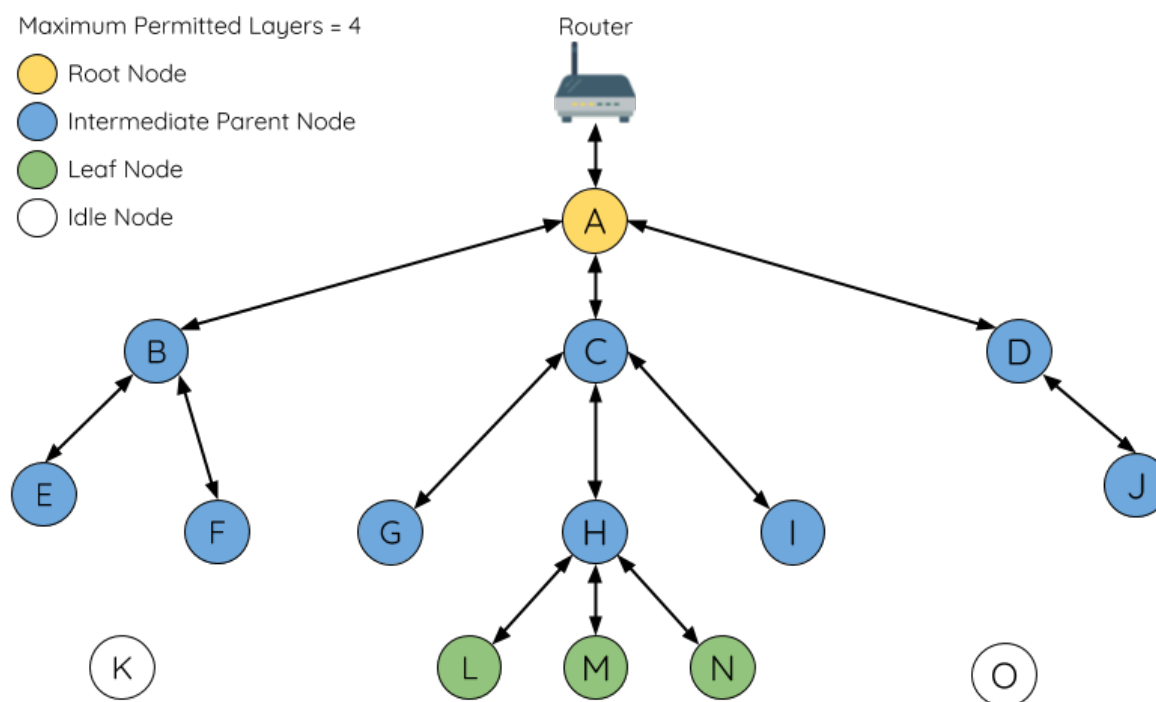


Fig. 19: ESP-MESH Node Types

- Current layer of Node
- Maximum number of layers permitted in the network
- Current number of child nodes
- Maximum number of downstream connections to accept

The signal strength of a potential upstream connection is represented by RSSI (Received Signal Strength Indication) of the beacon frames of the potential parent node. To prevent nodes from forming a weak upstream connection, ESP-MESH implements an RSSI threshold mechanism for beacon frames. If a node detects a beacon frame with an RSSI below a preconfigured threshold, the transmitting node will be disregarded when forming an upstream connection.

Panel A of the illustration above demonstrates how the RSSI threshold affects the number of parent node candidates an idle node has.

Panel B of the illustration above demonstrates how an RF shielding object can lower the RSSI of a potential parent node. Due to the RF shielding object, the area in which the RSSI of node X is above the threshold is significantly reduced. This causes the idle node to disregard node X even though node X is physically adjacent. The idle node will instead form an upstream connection with the physically distant node Y due to a stronger RSSI.

Note: Nodes technically still receive all beacon frames on the MAC layer. The RSSI threshold is an ESP-MESH feature that simply filters out all received beacon frames that are below the preconfigured threshold.

Preferred Parent Node

When an idle node has multiple parent nodes candidates (potential parent nodes), the idle node will form an upstream connection with the **preferred parent node**. The preferred parent node is determined based on the following criteria:

- Which layer the parent node candidate is situated on
- The number of downstream connections (child nodes) the parent node candidate currently has

The selection of the preferred parent node will always prioritize the parent node candidate on the shallowest layer of the network (including the root node). This helps minimize the total number of layers in an ESP-MESH network

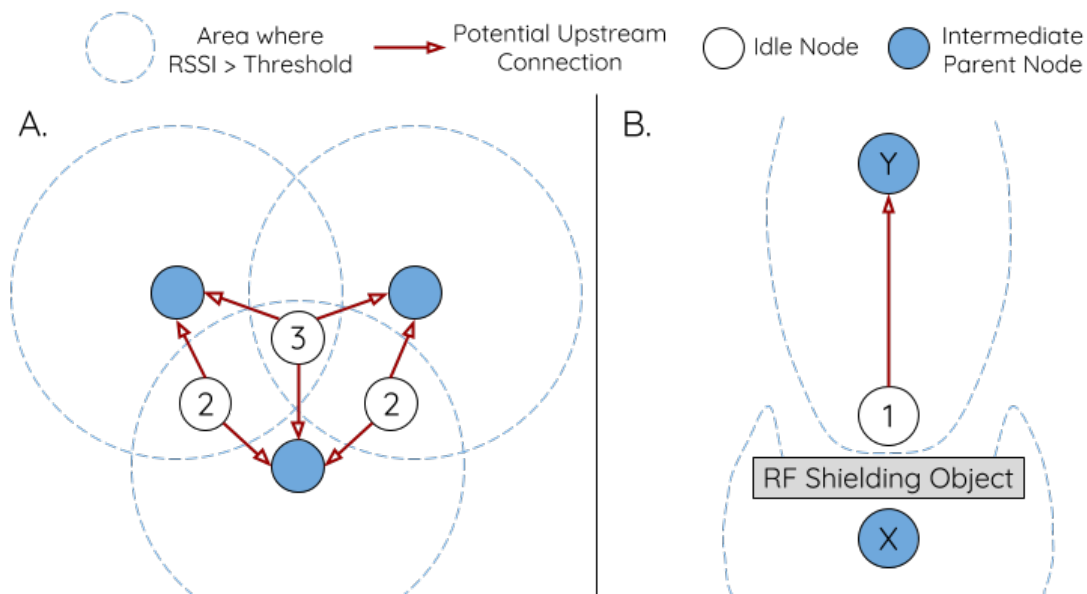


Fig. 20: Effects of RSSI Thresholding

when upstream connections are formed. For example, given a second layer node and a third layer node, the second layer node will always be preferred.

If there are multiple parent node candidates within the same layer, the parent node candidate with the least child nodes will be preferred. This criteria has the effect of balancing the number of downstream connections amongst nodes of the same layer.

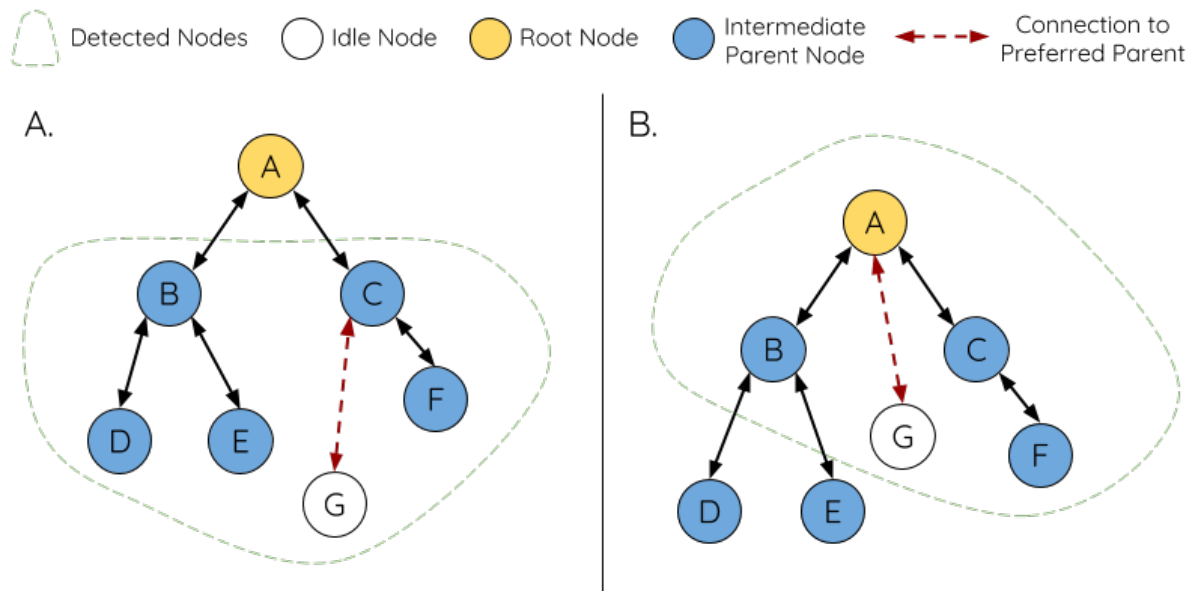


Fig. 21: Preferred Parent Node Selection

Panel A of the illustration above demonstrates an example of how the idle node G selects a preferred parent node given the five parent node candidates B/C/D/E/F. Nodes on the shallowest layer are preferred, hence nodes B/C are prioritized since they are second layer nodes whereas nodes D/E/F are on the third layer. Node C is selected as the preferred parent node due it having fewer downstream connections (fewer child nodes) compared to node B.

Panel B of the illustration above demonstrates the case where the root node is within range of the idle node G. In other words, the root node's beacon frames are above the RSSI threshold when received by node G. The root node is always the shallowest node in an ESP-MESH network hence is always the preferred parent given multiple

parent node candidates.

Note: Users may also define their own algorithm for selecting a preferred parent node, or force a node to only connect with a specific parent node (see the [Mesh Manual Networking Example](#)).

Routing Tables

Each node within an ESP-MESH network will maintain its individual routing table used to correctly route ESP-MESH packets (see [ESP-MESH Packet](#)) to the correct destination node. The routing table of a particular node will **consist of the MAC addresses of all nodes within the particular node's subnetwork** (including the MAC address of the particular node itself). Each routing table is internally partitioned into multiple subtables with each subtable corresponding to the subnetwork of each child node.

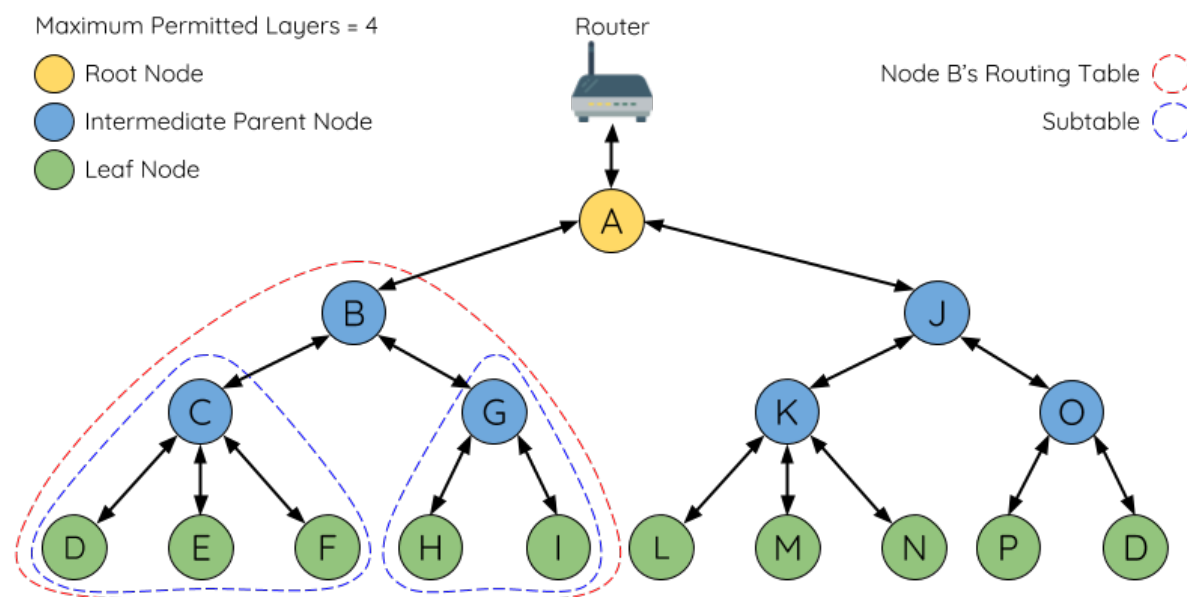


Fig. 22: ESP-MESH Routing Tables Example

Using the diagram above as an example, the routing table of node B would consist of the MAC addresses of nodes B to I (i.e. equivalent to the subnetwork of node B). Node B's routing table is internally partitioned into two subtables containing of nodes C to F and nodes G to I (i.e. equivalent to the subnetworks of nodes C and G respectively).

ESP-MESH utilizes routing tables to determine whether an ESP-MESH packet should be forwarded upstream or downstream based on the following rules.

1. If the packet's destination MAC address is within the current node's routing table and is not the current node, select the subtable that contains the destination MAC address and forward the data packet downstream to the child node corresponding to the subtable.
2. If the destination MAC address is not within the current node's routing table, forward the data packet upstream to the current node's parent node. Doing so repeatedly will result in the packet arriving at the root node where the routing table should contain all nodes within the network.

Note: Users can call `esp_mesh_get_routing_table()` to obtain a node's routing table, or `esp_mesh_get_routing_table_size()` to obtain the size of a node's routing table.

`esp_mesh_get_subnet_nodes_list()` can be used to obtain the corresponding subtable of a specific child node. Likewise `esp_mesh_get_subnet_nodes_num()` can be used to obtain the size of the subtable.

5.10.4 Building a Network

General Process

Warning: Before the ESP-MESH network building process can begin, certain parts of the configuration must be uniform across each node in the network (see `mesh_cfg_t`). Each node must be configured with **the same Mesh Network ID, router configuration, and softAP configuration.**

An ESP-MESH network building process involves selecting a root node, then forming downstream connections layer by layer until all nodes have joined the network. The exact layout of the network can be dependent on factors such as root node selection, parent node selection, and asynchronous power-on reset. However, the ESP-MESH network building process can be generalized into the following steps:

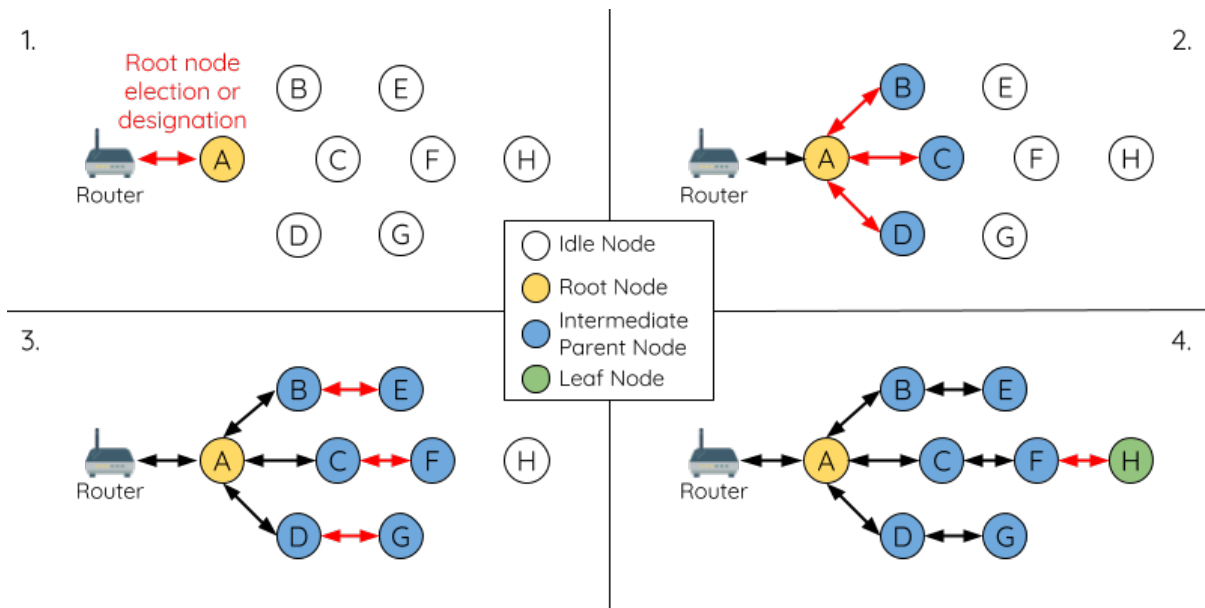


Fig. 23: ESP-MESH Network Building Process

1. Root Node Selection The root node can be designated during configuration (see section on *User Designated Root Node*), or dynamically elected based on the signal strength between each node and the router (see *Automatic Root Node Selection*). Once selected, the root node will connect with the router and begin allowing downstream connections to form. Referring to the figure above, node A is selected to be the root node hence node A forms an upstream connection with the router.

2. Second Layer Formation Once the root node has connected to the router, idle nodes in range of the root node will begin connecting with the root node thereby forming the second layer of the network. Once connected, the second layer nodes become intermediate parent nodes (assuming maximum permitted layers > 2) hence the next layer to form. Referring to the figure above, nodes B to D are in range of the root node. Therefore nodes B to D form upstream connections with the root node and become intermediate parent nodes.

3. Formation of remaining layers The remaining idle nodes will connect with intermediate parent nodes within range thereby forming a new layer in the network. Once connected, the idles nodes become intermediate parent node or leaf nodes depending on the networks maximum permitted layers. This step is repeated until there are no more idle nodes within the network or until the maximum permitted layer of the network has been reached. Referring to the figure above, nodes E/F/G connect with nodes B/C/D respectively and become intermediate parent nodes themselves.

4. Limiting Tree Depth To prevent the network from exceeding the maximum permitted number of layers, nodes on the maximum layer will automatically become leaf nodes once connected. This prevents any other idle node from connecting with the leaf node thereby prevent a new layer form forming. However if an idle node has no other potential parent node, it will remain idle indefinitely. Referring to the figure above, the network 's maximum permitted layers is set to four. Therefore when node H connects, it becomes a leaf node to prevent any downstream connections from forming.

Automatic Root Node Selection

The automatic selection of a root node involves an election process amongst all idle nodes based on their signal strengths with the router. Each idle node will transmit their MAC addresses and router RSSI values via Wi-Fi beacon frames. **The MAC address is used to uniquely identify each node in the network** whilst the **router RSSI** is used to indicate a node 's signal strength with reference to the router.

Each node will then simultaneously scan for the beacon frames from other idle nodes. If a node detects a beacon frame with a stronger router RSSI, the node will begin transmitting the contents of that beacon frame (i.e. voting for the node with the stronger router RSSI). The process of transmission and scanning will repeat for a preconfigured minimum number of iterations (10 iterations by default) and result in the beacon frame with the strongest router RSSI being propagated throughout the network.

After all iterations, each node will individually check for its **vote percentage** (number of votes/number of nodes participating in election) to determine if it should become the root node. **If a node has a vote percentage larger than a preconfigured threshold (90% by default), the node will become a root node.**

The following diagram demonstrates how an ESP-MESH network is built when the root node is automatically selected.

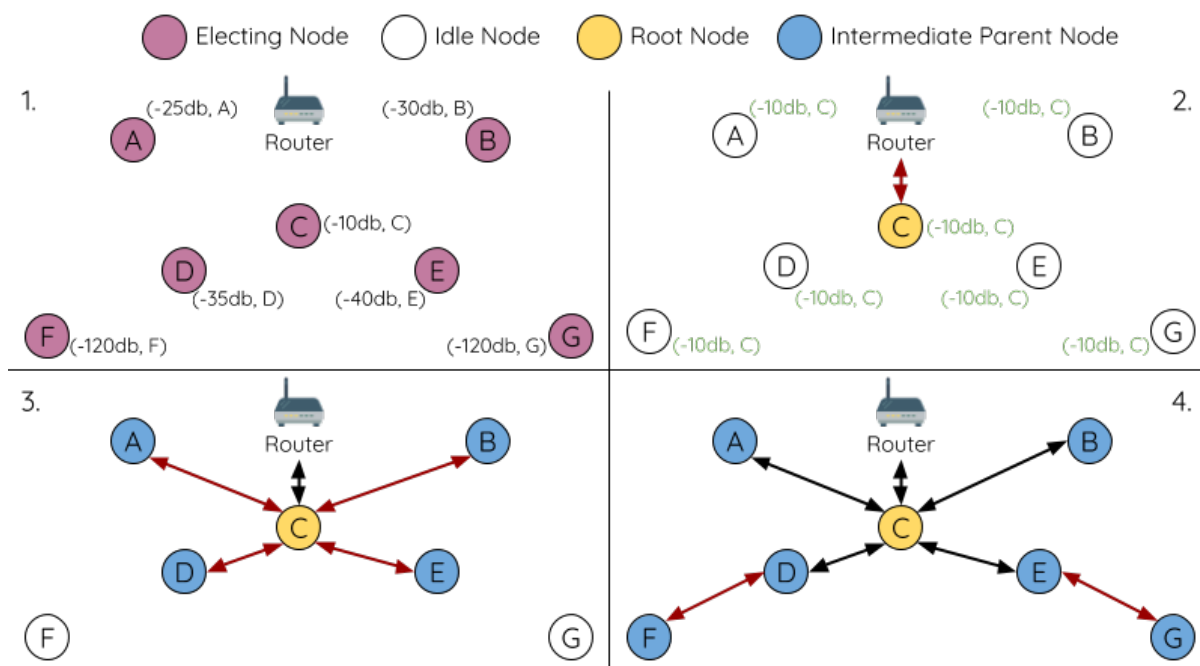


Fig. 24: Root Node Election Example

1. On power-on reset, each node begins transmitting beacon frames consisting of their own MAC addresses and their router RSSIs.
2. Over multiple iterations of transmission and scanning, the beacon frame with the strongest router RSSI is propagated throughout the network. Node C has the strongest router RSSI (-10db) hence its beacon frame is propagated throughout the network. All nodes participating in the election vote for node C thus giving node C a vote percentage of 100%. Therefore node C becomes a root node and connects with the router.
3. Once Node C has connected with the router, nodes A/B/D/E connect with node C as it is the preferred parent node (i.e. the shallowest node). Nodes A/B/D/E form the second layer of the network.

4. Node F and G connect with nodes D and E respectively and the network building process is complete.

Note: The minimum number of iterations for the election process can be configured using `esp_mesh_set_attempts()`. Users should adjust the number of iterations based on the number of nodes within the network (i.e. the larger the network the larger number of scan iterations required).

Warning: `Vote percentage threshold` can also be configured using `esp_mesh_set_vote_percentage()`. Setting a low vote percentage threshold **can result in two or more nodes becoming root nodes** within the same ESP-MESH network leading to the building of multiple networks. If such is the case, ESP-MESH has internal mechanisms to autonomously resolve the **root node conflict**. The networks of the multiple root nodes will be combined into a single network with a single root node. However, root node conflicts where two or more root nodes have the same router SSID but different router BSSID are not handled.

User Designated Root Node

The root node can also be designated by user which will entail the designated root node to directly connect with the router and forgo the election process. When a root node is designated, all other nodes within the network must also forgo the election process to prevent the occurrence of a root node conflict. The following diagram demonstrates how an ESP-MESH network is built when the root node is designated by the user.

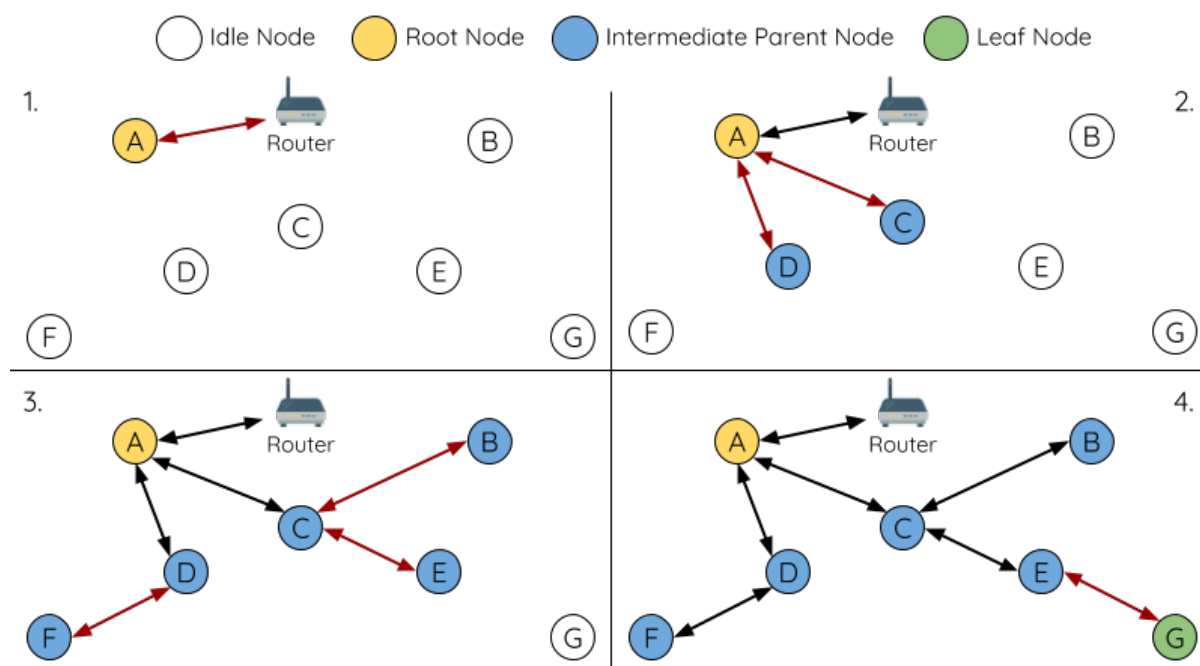


Fig. 25: Root Node Designation Example (Root Node = A, Max Layers = 4)

1. Node A is designated the root node by the user therefore directly connects with the router. All other nodes forgo the election process.
2. Nodes C/D connect with node A as their preferred parent node. Both nodes form the second layer of the network.
3. Likewise, nodes B/E connect with node C, and node F connects with node D. Nodes B/E/F form the third layer of the network.
4. Node G connects with node E, forming the fourth layer of the network. However the maximum permitted number of layers in this network is configured as four, therefore node G becomes a leaf node to prevent any new layers from forming.

Note: When designating a root node, the root node should call `esp_mesh_set_parent()` in order to directly connect with the router. Likewise, all other nodes should call `esp_mesh_fix_root()` to forgo the election process.

Parent Node Selection

By default, ESP-MESH is self organizing meaning that each node will autonomously select which potential parent node to form an upstream connection with. The autonomously selected parent node is known as the preferred parent node. The criteria used for selecting the preferred parent node is designed to reduce the number of layers in the ESP-MESH network and to balance the number of downstream connections between potential parent nodes (see section on [Preferred Parent Node](#)).

However ESP-MESH also allows users to disable self-organizing behavior which will allow users to define their own criteria for parent node selection, or to configure nodes to have designated parent nodes (see the [Mesh Manual Networking Example](#)).

Asynchronous Power-on Reset

ESP-MESH network building can be affected by the order in which nodes power-on. If certain nodes within the network power-on asynchronously (i.e. separated by several minutes), **the final structure of the network could differ from the ideal case where all nodes are powered on synchronously**. Nodes that are delayed in powering on will adhere to the following rules:

Rule 1: If a root node already exists in the network, the delayed node will not attempt to elect a new root node, even if it has a stronger RSSI with the router. The delayed node will instead join the network like any other idle node by connecting with a preferred parent node. If the delayed node is the designated root node, all other nodes in the network will remain idle until the delayed node powers-on.

Rule 2: If a delayed node forms an upstream connection and becomes an intermediate parent node, it may also become the new preferred parent of other nodes (i.e. being a shallower node). This will cause the other nodes to switch their upstream connections to connect with the delayed node (see [Parent Node Switching](#)).

Rule 3: If an idle node has a designated parent node which is delayed in powering-on, the idle node will not attempt to form any upstream connections in the absence of its designated parent node. The idle node will remain idle indefinitely until its designated parent node powers-on.

The following example demonstrates the effects of asynchronous power-on with regards to network building.

1. Nodes A/C/D/F/G/H are powered-on synchronously and begin the root node election process by broadcasting their MAC addresses and router RSSIs. Node A is elected as the root node as it has the strongest RSSI.
2. Once node A becomes the root node, the remaining nodes begin forming upstream connections layer by layer with their preferred parent nodes. The result is a network with five layers.
3. Node B/E are delayed in powering-on but neither attempt to become the root node even though they have stronger router RSSIs (-20db and -10db) compared to node A. Instead both delayed nodes form upstream connections with their preferred parent nodes A and C respectively. Both Nodes B/E become intermediate parent nodes after connecting.
4. Nodes D/G switch their upstream connections as node B is the new preferred parent node due to it being on a shallower layer (second layer node). Due to the switch, the resultant network has three layers instead of the original five layers.

Synchronous Power-On: Had all nodes powered-on synchronously, node E would have become the root node as it has the strongest router RSSI (-10db). This would result in a significantly different network layout compared to the network formed under the conditions of asynchronous power-on. **However the synchronous power-on network layout can still be reached if the user manually switches the root node** (see `esp_mesh_waive_root()`).

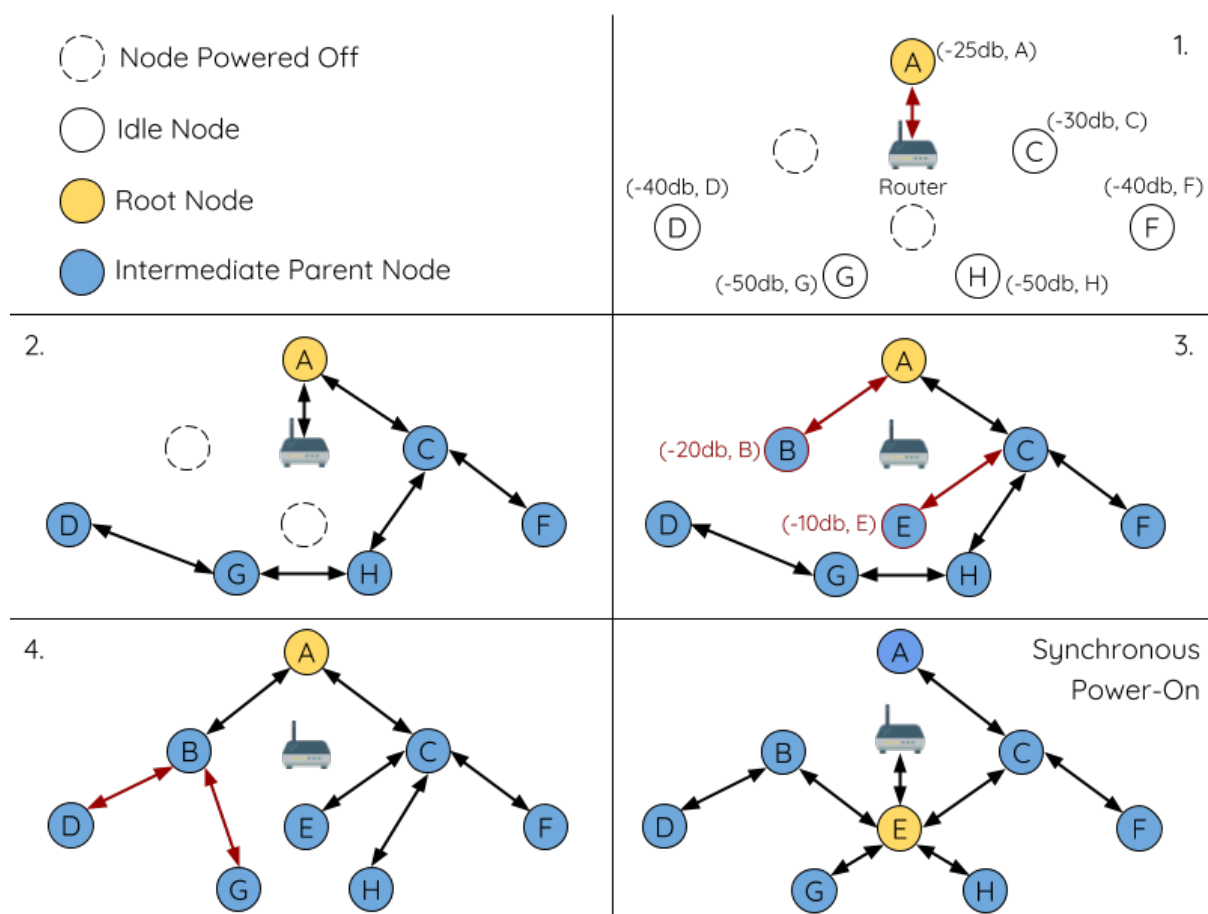


Fig. 26: Network Building with Asynchronous Power On Example

Note: Differences in parent node selection caused by asynchronous power-on are autonomously corrected for to some extent in ESP-MESH (see *Parent Node Switching*)

Loop-back Avoidance, Detection, and Handling

A loop-back is the situation where a particular node forms an upstream connection with one of its descendant nodes (a node within the particular node's subnetwork). This results in a circular connection path thereby breaking the tree topology. ESP-MESH prevents loop-back during parent selection by excluding nodes already present in the selecting node's routing table (see *Routing Tables*) thus prevents a particular node from attempting to connect to any node within its subnetwork.

In the event that a loop-back occurs, ESP-MESH utilizes a path verification mechanism and energy transfer mechanism to detect the loop-back occurrence. The parent node of the upstream connection that caused the loop-back will then inform the child node of the loop-back and initiate a disconnection.

5.10.5 Managing a Network

ESP-MESH is a self healing network meaning it can detect and correct for failures in network routing. Failures occur when a parent node with one or more child nodes breaks down, or when the connection between a parent node and its child nodes becomes unstable. Child nodes in ESP-MESH will autonomously select a new parent node and form an upstream connection with it to maintain network interconnectivity. ESP-MESH can handle both Root Node Failures and Intermediate Parent Node Failures.

Root Node Failure

If the root node breaks down, the nodes connected with it (second layer nodes) will promptly detect the failure of the root node. The second layer nodes will initially attempt to reconnect with the root node. However after multiple failed attempts, the second layer nodes will initialize a new round of root node election. **The second layer node with the strongest router RSSI will be elected as the new root node** whilst the remaining second layer nodes will form an upstream connection with the new root node (or a neighboring parent node if not in range).

If the root node and multiple downstream layers simultaneously break down (e.g. root node, second layer, and third layer), the shallowest layer that is still functioning will initialize the root node election. The following example illustrates an example of self healing from a root node break down.

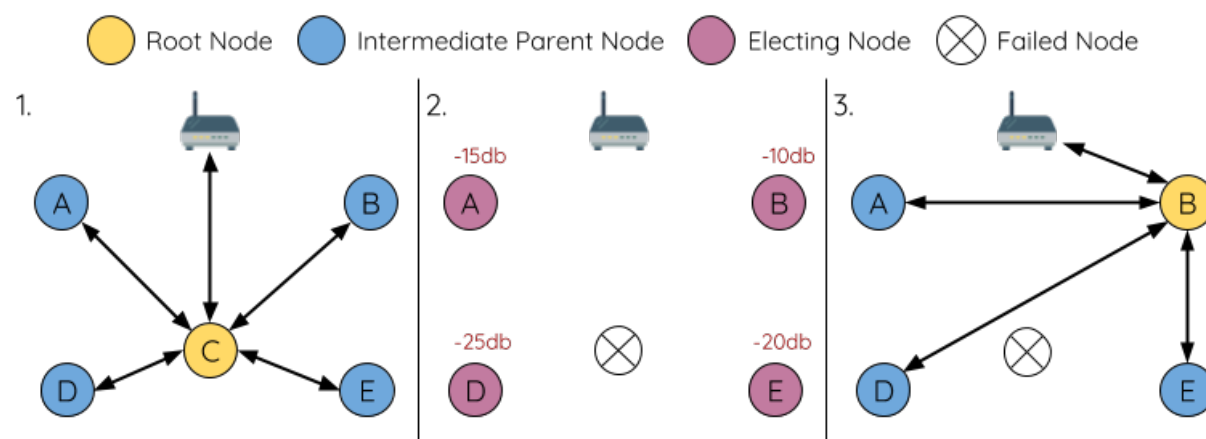


Fig. 27: Self Healing From Root Node Failure

- Node C is the root node of the network. Nodes A/B/D/E are second layer nodes connected to node C.
- Node C breaks down. After multiple failed attempts to reconnect, the second layer nodes begin the election process by broadcasting their router RSSIs. Node B has the strongest router RSSI.

3. Node B is elected as the root node and begins accepting downstream connections. The remaining second layer nodes A/D/E form upstream connections with node B thus the network is healed and can continue operating normally.

Note: If a designated root node breaks down, the remaining nodes **will not autonomously attempt to elect a new root node** as an election process will never be attempted whilst a designated root node is used.

Intermediate Parent Node Failure

If an intermediate parent node breaks down, the disconnected child nodes will initially attempt to reconnect with the parent node. After multiple failed attempts to reconnect, each child node will begin to scan for potential parent nodes (see *Beacon Frames & RSSI Thresholding*).

If other potential parent nodes are available, each child node will individually select a new preferred parent node (see *Preferred Parent Node*) and form an upstream connection with it. If there are no other potential parent nodes for a particular child node, it will remain idle indefinitely.

The following diagram illustrates an example of self healing from an Intermediate Parent Node break down.

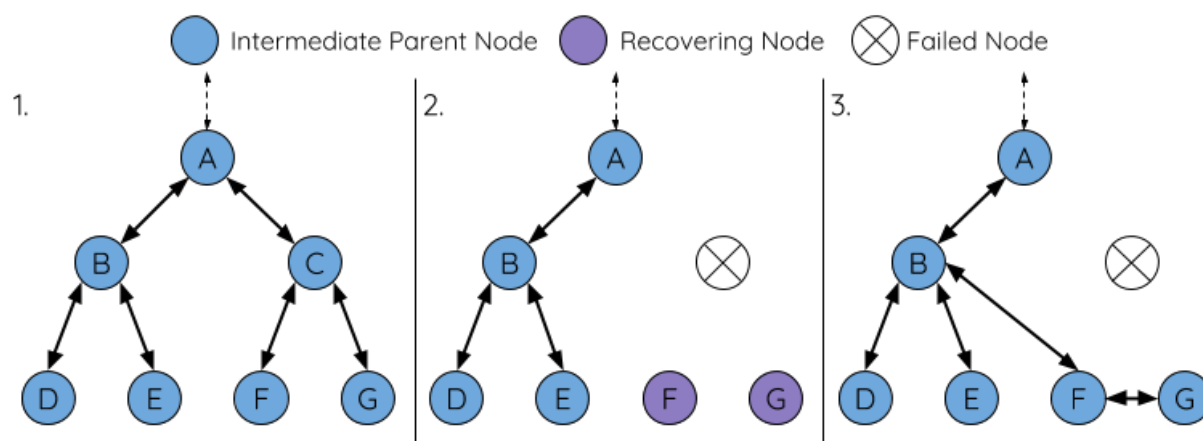


Fig. 28: Self Healing From Intermediate Parent Node Failure

1. The following branch of the network consists of nodes A to G.
2. Node C breaks down. Nodes F/G detect the break down and attempt to reconnect with node C. After multiple failed attempts to reconnect, nodes F/G begin to select a new preferred parent node.
3. Node G is out of range from any other parent node hence remains idle for the time being. Node F is in range of nodes B/E, however node B is selected as it is the shallower node. Node F becomes an intermediate parent node after connecting with Node B thus node G can connect with node F. The network is healed, however the network routing as been affected and an extra layer has been added.

Note: If a child node has a designated parent node that breaks down, the child node will make no attempt to connect with a new parent node. The child node will remain idle indefinitely.

Root Node Switching

ESP-MESH does not automatically switch the root node unless the root node breaks down. Even if the root node's router RSSI degrades to the point of disconnection, the root node will remain unchanged. Root node switching is the act of explicitly starting a new election such that a node with a stronger router RSSI will be elected as the new root node. This can be a useful method of adapting to degrading root node performance.

To trigger a root node switch, the current root node must explicitly call `esp_mesh_waive_root()` to trigger a new election. The current root node will signal all nodes within the network to begin transmitting and scanning for beacon frames (see [Automatic Root Node Selection](#)) **whilst remaining connected to the network (i.e. not idle)**. If another node receives more votes than the current root node, a root node switch will be initiated. **The root node will remain unchanged otherwise.**

A newly elected root node sends a **switch request** to the current root node which in turn will respond with an acknowledgment signifying both nodes are ready to switch. Once the acknowledgment is received, the newly elected root node will disconnect from its parent and promptly form an upstream connection with the router thereby becoming the new root node of the network. The previous root node will disconnect from the router **whilst maintaining all of its downstream connections** and enter the idle state. The previous root node will then begin scanning for potential parent nodes and selecting a preferred parent.

The following diagram illustrates an example of a root node switch.

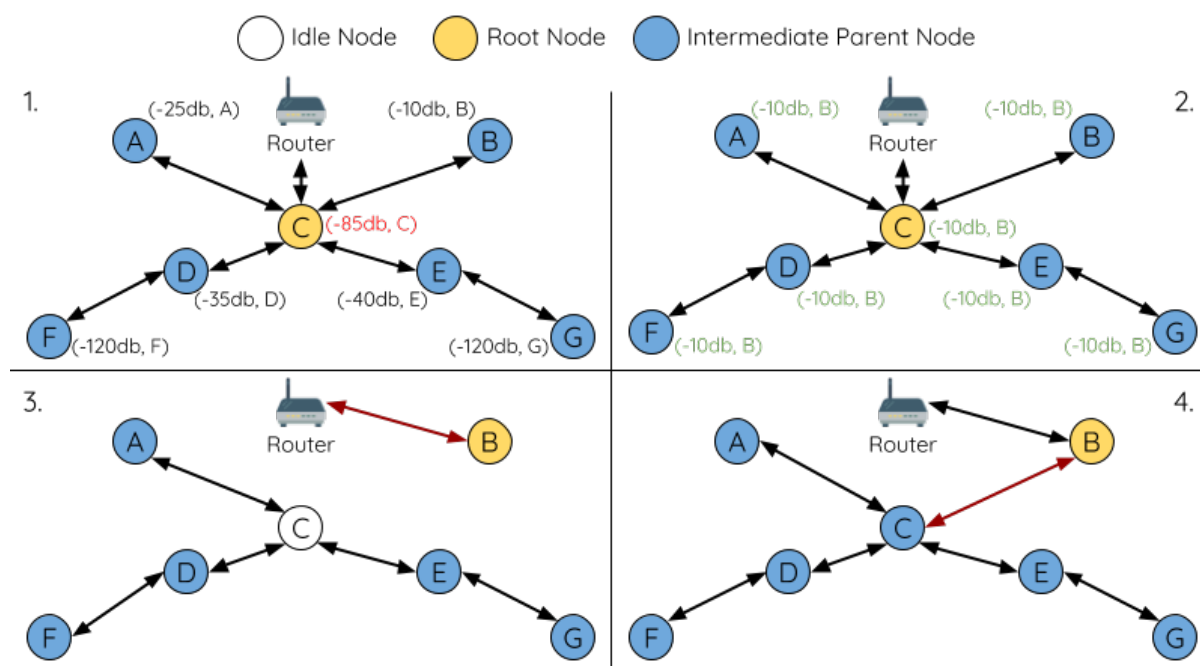


Fig. 29: Root Node Switch Example

1. Node C is the current root node but has degraded signal strength with the router (-85db). The node C triggers a new election and all nodes begin transmitting and scanning for beacon frames **whilst still being connected**.
2. After multiple rounds of transmission and scanning, node B is elected as the new root node. Node B sends node C a **switch request** and node C responds with an acknowledgment.
3. Node B disconnects from its parent and connects with the router becoming the networks new root node. Node C disconnects from the router, enters the idle state, and begins scanning for and selecting a new preferred parent node. **Node C maintains all its downstream connections throughout this process.**
4. Node C selects node B as its preferred parent node, forms an upstream connection, and becomes a second layer node. The network layout is similar after the switch as node C still maintains the same subnetwork. However each node in node C's subnetwork has been placed one layer deeper as a result of the switch. [Parent Node Switching](#) may adjust the network layout afterwards if any nodes have a new preferred parent node as a result of the root node switch.

Note: Root node switching must require an election hence is only supported when using a self-organized ESP-MESH network. In other words, root node switching cannot occur if a designated root node is used.

Parent Node Switching

Parent Node Switching entails a child node switching its upstream connection to another parent node of a shallower layer. **Parent Node Switching occurs autonomously** meaning that a child node will change its upstream connection automatically if a potential parent node of a shallower layer becomes available (i.e. due to a *Asynchronous Power-on Reset*).

All potential parent nodes periodically transmit beacon frames (see *Beacon Frames & RSSI Thresholding*) allowing for a child node to scan for the availability of a shallower parent node. Due to parent node switching, a self-organized ESP-MESH network can dynamically adjust its network layout to ensure each connection has a good RSSI and that the number of layers in the network is minimized.

5.10.6 Data Transmission

ESP-MESH Packet

ESP-MESH network data transmissions use ESP-MESH packets. ESP-MESH packets are **entirely contained within the frame body of a Wi-Fi data frame**. A multi-hop data transmission in an ESP-MESH network will involve a single ESP-MESH packet being carried over each wireless hop by a different Wi-Fi data frame.

The following diagram shows the structure of an ESP-MESH packet and its relation with a Wi-Fi data frame.

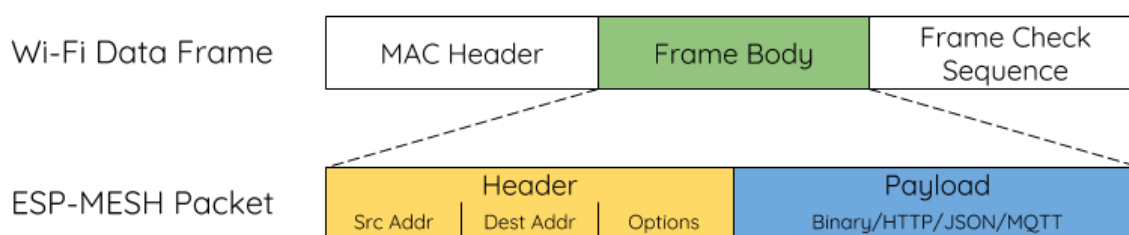


Fig. 30: ESP-MESH Packet

The header of an ESP-MESH packet contains the MAC addresses of the source and destination nodes. The options field contains information pertaining to the special types of ESP-MESH packets such as a group transmission or a packet originating from the external IP network (see *MESH_OPT_SEND_GROUP* and *MESH_OPT_RECV_DS_ADDR*).

The payload of an ESP-MESH packet contains the actual application data. This data can be raw binary data, or encoded under an application layer protocol such as HTTP, MQTT, and JSON (see *mesh_proto_t*).

Note: When sending an ESP-MESH packet to the external IP network, the destination address field of the header will contain the IP address and port of the target server rather than the MAC address of a node (see *mesh_addr_t*). Furthermore the root node will handle the formation of the outgoing TCP/IP packet.

Group Control & Multicasting

Multicasting is a feature that allows a single ESP-MESH packet to be transmitted simultaneously to multiple nodes within the network. Multicasting in ESP-MESH can be achieved by either specifying a list of target nodes, or specifying a preconfigured group of nodes. Both methods of multicasting are called via *esp_mesh_send()*.

To multicast by specifying a list of target nodes, users must first set the ESP-MESH packet's destination address to the **Multicast-Group Address** (01:00:5E:xx:xx:xx). This signifies that the ESP-MESH packet is a multicast packet with a group of addresses, and that the address should be obtained from the header options. Users must then list the MAC addresses of the target nodes as options (see *mesh_opt_t* and *MESH_OPT_SEND_GROUP*). This

method of multicasting requires no prior setup but can incur a large amount of overhead data as each target node's MAC address must be listed in the options field of the header.

Multicasting by group allows a ESP-MESH packet to be transmitted to a preconfigured group of nodes. Each grouping is identified by a unique ID, and a node can be placed into a group via `esp_mesh_set_group_id()`. Multicasting to a group involves setting the destination address of the ESP-MESH packet to the target group ID. Furthermore, the `MESH_DATA_GROUP` flag must set. Using groups to multicast incurs less overhead, but requires nodes to previously added into groups.

Note: During a multicast, all nodes within the network still receive the ESP-MESH packet on the MAC layer. However, nodes not included in the MAC address list or the target group will simply filter out the packet.

Broadcasting

Broadcasting is a feature that allows a single ESP-MESH packet to be transmitted simultaneously to all nodes within the network. Each node essentially forwards a broadcast packet to all of its upstream and downstream connections such that the packet propagates throughout the network as quickly as possible. However, ESP-MESH utilizes the following methods to avoid wasting bandwidth during a broadcast.

1. When an intermediate parent node receives a broadcast packet from its parent, it will forward the packet to each of its child nodes whilst storing a copy of the packet for itself.
2. When an intermediate parent node is the source node of the broadcast, it will transmit the broadcast packet upstream to its parent node and downstream to each of its child nodes.
3. When an intermediate parent node receives a broadcast packet from one of its child nodes, it will forward the packet to its parent node and each of its remaining child nodes whilst storing a copy of the packet for itself.
4. When a leaf node is the source node of a broadcast, it will directly transmit the packet to its parent node.
5. When the root node is the source node of a broadcast, the root node will transmit the packet to all of its child nodes.
6. When the root node receives a broadcast packet from one of its child nodes, it will forward the packet to each of its remaining child nodes whilst storing a copy of the packet for itself.
7. When a node receives a broadcast packet with a source address matching its own MAC address, the node will discard the broadcast packet.
8. When an intermediate parent node receives a broadcast packet from its parent node which was originally transmitted from one of its child nodes, it will discard the broadcast packet

Upstream Flow Control

ESP-MESH relies on parent nodes to control the upstream data flow of their immediate child nodes. To prevent a parent node's message buffer from overflowing due to an overload of upstream transmissions, a parent node will allocate a quota for upstream transmissions known as a **receiving window** for each of its child nodes. **Each child node must apply for a receiving window before it is permitted to transmit upstream.** The size of a receiving window can be dynamically adjusted. An upstream transmission from a child node to the parent node consists of the following steps:

1. Before each transmission, the child node sends a window request to its parent node. The window request consists of a sequence number which corresponds to the child node's data packet that is pending transmission.
2. The parent node receives the window request and compares the sequence number with the sequence number of the previous packet sent by the child node. The comparison is used to calculate the size of the receiving window which is transmitted back to the child node.
3. The child node transmits the data packet in accordance with the window size specified by the parent node. If the child node depletes its receiving window, it must obtain another receiving windows by sending a request before it is permitted to continue transmitting.

Note: ESP-MESH does not support any downstream flow control.

Warning: Due to *Parent Node Switching*, packet loss may occur during upstream transmissions.

Due to the fact that the root node acts as the sole interface to an external IP network, it is critical that downstream nodes are aware of the root node's connection status with the external IP network. Failing to do so can lead to nodes attempting to pass data upstream to the root node whilst it is disconnected from the IP network. This results in unnecessary transmissions and packet loss. ESP-MESH address this issue by providing a mechanism to stabilize the throughput of outgoing data based on the connection status between the root node and the external IP network. The root node can broadcast its external IP network connection status to all other nodes by calling `esp_mesh_post_toDS_state()`.

Bi-Directional Data Stream

The following diagram illustrates the various network layers involved in an ESP-MESH Bidirectional Data Stream.

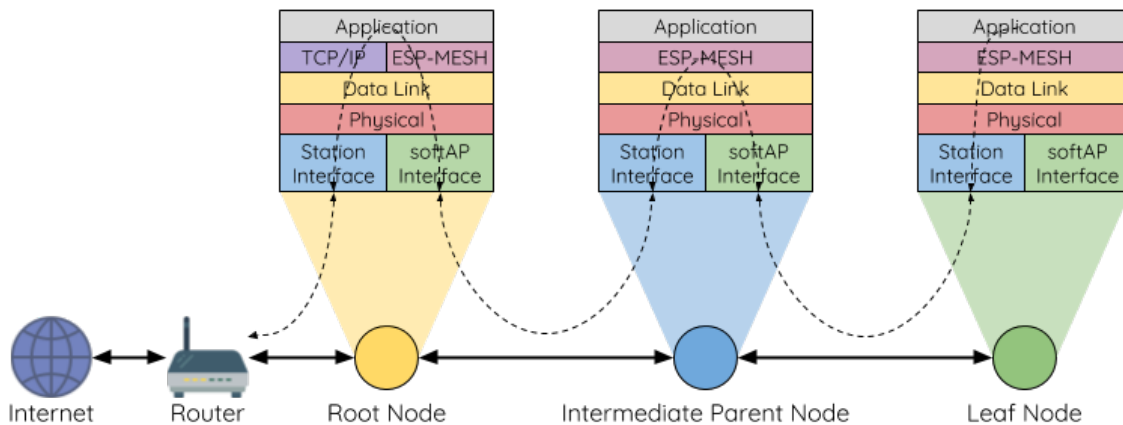


Fig. 31: ESP-MESH Bidirectional Data Stream

Due to the use of *Routing Tables*, **ESP-MESH is able to handle pack forwarding entirely on the mesh layer**. A TCP/IP layer is only required on the root node when it transmits/receives a packet to/from an external IP network.

5.10.7 Channel Switching

Background

In traditional Wi-Fi networks, **channels** are predetermined frequency ranges. In an infrastructure basic service set (BSS), the serving AP and its connected stations must be on the same operating channels (1 to 14) in which beacons are transmitted. Physically adjacent BSS (Basic Service Sets) operating on the same channel can lead to interference and degraded performance.

In order to allow a BSS adapt to changing physical layer conditions and maintain performance, Wi-Fi contains mechanisms for **network channel switching**. A network channel switch is an attempt to move a BSS to a new operating channel whilst minimizing disruption to the BSS during this process. However it should be recognized that a channel switch may be unsuccessful in moving all stations to the new operating channel.

In an infrastructure Wi-Fi network, network channel switches are triggered by the AP with the aim of having the AP and all connected stations synchronously switch to a new channel. Network channel switching is implemented by embedding a **Channel Switch Announcement (CSA)** element within the AP's periodically transmitted beacon

frames. The CSA element is used to advertise to all connected stations regarding an upcoming network channel switch and will be included in multiple beacon frames up until the switch occurs.

A CSA element contains information regarding the **New Channel Number** and a **Channel Switch Count** which indicates the number of beacon frame intervals (TBTTs) remaining until the network channel switch occurs. Therefore, the Channel Switch Count is decremented every beacon frame and allows connected stations to synchronize their channel switch with the AP.

ESP-MESH Network Channel Switching

ESP-MESH Network Channel Switching also utilize beacon frames that contain a CSA element. However, being a multi-hop network makes the switching process in ESP-MESH is more complex due to the fact that a beacon frame might not be able to reach all nodes within the network (i.e. in a single hop). Therefore, an ESP-MESH network relies on nodes to forward the CSA element so that it is propagated throughout the network.

When an intermediate parent node with one or more child nodes receives a beacon frame containing a CSA, the node will forward the CSA element by including the element in its next transmitted beacon frame (i.e. with the same **New Channel Number** and **Channel Switch Count**). Given that all nodes within an ESP-MESH network receive the same CSA, the nodes can synchronize their channel switches using the Channel Switch Count, albeit with a short delay due to CSA element forwarding.

An ESP-MESH network channel switch can be triggered by either the router or the root node.

Root Node Triggered A root node triggered channel switch can only occur when the ESP-MESH network is not connected to a router. By calling `esp_mesh_switch_channel()`, the root node will set an initial Channel Switch Count value and begin including a CSA element in its beacon frames. Each CSA element is then received by second layer nodes, and forwarded downstream in the their own beacon frames.

Router Triggered When an ESP-MESH network is connected to a router, the entire network must use the same channel as the router. Therefore, **the root node will not be permitted to trigger a channel switch when it is connected to a router.**

When the root node receives beacon frame containing a CSA element from the router, **the root node will set Channel Switch Count value in the CSA element to a custom value before forwarding it downstream via beacon frames.** It will also decrement the Channel Switch Count of subsequent CSA elements relative to the custom value. This custom value can be based on factors such as the number of network layers, the current number of nodes etc.

The setting the Channel Switch Count value to a custom value is due to the fact that the ESP-MESH network and its router may have a different and varying beacon intervals. Therefore, the Channel Switch Count value provided by the router is irrelevant to an ESP-MESH network. By using a custom value, nodes within the ESP-MESH network are able to switch channels synchronously relative to the ESP-MESH network's beacon interval. However, this will also result in the ESP-MESH network's channel switch being unsynchronized with the channel switch of the router and its connected stations.

Impact of Network Channel Switching

- **Due to the ESP-MESH network channel switch being unsynchronized with the router's channel switch, there will be**
 - The ESP-MESH network's channel switch time is dependent on the ESP-MESH network's beacon interval and the root node's custom Channel Switch Count value.
 - The channel discrepancy prevents any data exchange between the root node and the router during that ESP-MESH network's switch.
 - In the ESP-MESH network, the root node and intermediate parent nodes will request their connected child nodes to stop transmissions until the channel switch takes place by setting the **Channel Switch Mode** field in the CSA element to 1.
 - Frequent router triggered network channel switches can degrade the ESP-MESH network's performance. Note that this can be caused by the ESP-MESH network itself (e.g. due to wireless medium

contention with ESP-MESH network). If this is the case, users should disable the automatic channel switching on the router and use a specified channel instead.

- **When there is a temporary channel discrepancy, the root node remains technically connected to the router.**
 - Disconnection occurs after the root node fails to receive any beacon frames or probe responses from the router over a fixed number of router beacon intervals.
 - Upon disconnection, the root node will automatically re-scan all channels for the presence of a router.
- **If the root node is unable to receive any of the router's CSA beacon frames (e.g. due to short switch time given by the router).**
 - After the router switches channels, the root node will no longer be able to receive the router's beacon frames and probe responses and result in a disconnection after a fixed number of beacon intervals.
 - The root node will re-scan all channels for the router after disconnection.
 - The root node will maintain downstream connections throughout this process.

Note: Although ESP-MESH network channel switching aims to move all nodes within the network to a new operating channel, it should be recognized that a channel switch might not successfully move all nodes (e.g. due to reasons such as node failures).

Channel and Router Switching Configuration

ESP-MESH allows for autonomous channel switching to be enabled/disabled via configuration. Likewise, autonomous router switching (i.e. when a root node autonomously connects to another router) can also be enabled/disabled by configuration. Autonomous channel switching and router switching is dependent on the following configuration parameters and run-time conditions.

Allow Channel Switch: This parameter is set via the `allow_channel_switch` field of the `mesh_cfg_t` structure and permits an ESP-MESH network to dynamically switch channels when set.

Preset Channel: An ESP-MESH network can have a preset channel by setting the `channel` field of the `mesh_cfg_t` structure to the desired channel number. If this field is unset, the `allow_channel_switch` parameter is overridden such that channel switches are always permitted.

Allow Router Switch: This parameter is set via the `allow_router_switch` field of the `mesh_router_t` and permits an ESP-MESH to dynamically switch to a different router when set.

Preset Router BSSID: An ESP-MESH network can have a preset router by setting the `bssid` field of the `mesh_router_t` structure to the BSSID of the desired router. If this field is unset, the `allow_router_switch` parameter is overridden such that router switches are always permitted.

Root Node Present: The presence of a root node will also affect whether or a channel or router switch is permitted.

The following table illustrates how the different combinations of parameters/conditions affect whether channel switching and/or router switching is permitted. Note that X represents a “don't care” for the parameter.

| Configuration and Conditions | | | | | | Result | | | | | | |
|------------------------------|----------------------|---------------------|--------|---------------------|-------------------|--------------------|--------------|------------------|--------------|------------------|---|------------------|
| Preset Channel | Allow Channel Switch | Preset Router BSSID | Router | Allow Router Switch | Root Node Present | Permitted Switches | | | | | | |
| N | X | N | | X | X | Channel & Router | | | | | | |
| | | Y | | N | | Channel Only | | | | | | |
| | | Y | | Y | | Channel & Router | | | | | | |
| Y | Y | N | | X | X | Channel & Router | | | | | | |
| | | | | | | N | N | Router Only | | | | |
| | | | | | | N | Y | Channel & Router | | | | |
| | Y | Y | Y | | N | X | Channel Only | | | | | |
| | | | | | | | N | N | None | | | |
| | | | | | | | N | Y | Channel Only | | | |
| | | | | | Y | Y | | Y | X | Channel & Router | | |
| | | | | | | | | | | N | N | Router Only |
| | | | | | | | | | | N | Y | Channel & Router |

5.10.8 Performance

The performance of an ESP-MESH network can be evaluated based on multiple metrics such as the following:

Network Building Time: The amount of time taken to build an ESP-MESH network from scratch.

Healing Time: The amount of time taken for the network to detect a node break down and carry out appropriate actions to heal the network (such as generating a new root node or forming new connections).

Per-hop latency: The latency of data transmission over one wireless hop. In other words, the time taken to transmit a data packet from a parent node to a child node or vice versa.

Network Node Capacity: The total number of nodes the ESP-MESH network can simultaneously support. This number is determined by the maximum number of downstream connections a node can accept and the maximum number of layers permissible in the network.

The following table lists the common performance figures of an ESP-MESH network. However users should note that performance numbers can vary greatly between installations based on network configuration and operating environment.

| Function | Description |
|--------------------------|--|
| Networking Building Time | < 60 seconds |
| Healing time | Root Node Break Down: < 10 seconds Child Node Break Down: < 5 seconds |
| Per-hop latency | 10 to 30 milliseconds |

Note: The following test conditions were used to generate the performance figures above.

- Number of test devices: **100**
- Maximum Downstream Connections to Accept: **6**
- Maximum Permissible Layers: **6**

Note: Throughput depends on packet error rate and hop count.

Note: The throughput of root node's access to the external IP network is directly affected by the number of nodes in the ESP-MESH network and the bandwidth of the router.

5.10.9 Further Notes

- Data transmission uses Wi-Fi WPA2-PSK encryption
- Mesh networking IE uses AES encryption

Router and internet icon made by Smashicons from www.flaticon.com

5.11 ESP32 Core Dump

5.11.1 Overview

ESP-IDF provides support to generate core dumps on unrecoverable software errors. This useful technique allows post-mortem analysis of software state at the moment of failure. Upon the crash system enters panic state, prints some information and halts or reboots depending configuration. User can choose to generate core dump in order to analyse the reason of failure on PC later on. Core dump contains snapshots of all tasks in the system at the moment of failure. Snapshots include tasks control blocks (TCB) and stacks. So it is possible to find out what task, at what instruction (line of code) and what callstack of that task lead to the crash. ESP-IDF provides special script *espcoredump.py* to help users to retrieve and analyse core dumps. This tool provides two commands for core dumps analysis:

- `info_corefile` - prints crashed task's registers, callstack, list of available tasks in the system, memory regions and contents of memory stored in core dump (TCBs and stacks)
- `dbg_corefile` - creates core dump ELF file and runs GDB debug session with this file. User can examine memory, variables and tasks states manually. Note that since not all memory is saved in core dump only values of variables allocated on stack will be meaningful

5.11.2 Configuration

There are a number of core dump related configuration options which user can choose in project configuration menu (*idf.py menuconfig*).

1. Core dump data destination (*Components -> Core dump -> Data destination*):
 - Save core dump to Flash (Flash)
 - Print core dump to UART (UART)
 - Disable core dump generation (None)
2. Core dump data format (*Components -> Core dump -> Core dump data format*):
 - ELF format (Executable and Linkable Format file for core dump)
 - Binary format (Basic binary format for core dump)

The ELF format contains extended features and allow to save more information about broken tasks and crashed software but it requires more space in the flash memory. It also stores SHA256 of crashed application image. This format of core dump is recommended for new software designs and is flexible enough to extend saved information for future revisions. The Binary format is kept for compatibility standpoint, it uses less space in the memory to keep data and provides better performance.

3. Maximum number of tasks snapshots in core dump (*Components -> Core dump -> Maximum number of tasks*).
4. Delay before core dump is printed to UART (*Components -> Core dump -> Delay before print to UART*). Value is in ms.
5. Type of data integrity check for core dump (*Components -> Core dump -> Core dump data integrity check*).
 - Use CRC32 for core dump integrity verification
 - Use SHA256 for core dump integrity verification

The SHA256 hash algorithm provides greater probability of detecting corruption than a CRC32 with multiple bit errors. The CRC32 option provides better calculation performance and consumes less memory for storage.

5.11.3 Save core dump to flash

When this option is selected core dumps are saved to special partition on flash. When using default partition table files which are provided with ESP-IDF it automatically allocates necessary space on flash, But if user wants to use its own layout file together with core dump feature it should define separate partition for core dump as it is shown below:

```
# Name, Type, SubType, Offset, Size
# Note: if you have increased the bootloader size, make sure to update the offsets.
↳to avoid overlap
nvs, data, nvs, 0x9000, 0x6000
phy_init, data, phy, 0xF000, 0x1000
factory, app, factory, 0x10000, 1M
coredump, data, coredump,, 64K
```

There are no special requirements for partition name. It can be chosen according to the user application needs, but partition type should be 'data' and sub-type should be 'coredump'. Also when choosing partition size note that core dump data structure introduces constant overhead of 20 bytes and per-task overhead of 12 bytes. This overhead does not include size of TCB and stack for every task. So partition size should be at least 20 + max tasks number x (12 + TCB size + max task stack size) bytes.

The example of generic command to analyze core dump from flash is: `espcoredump.py -p </path/to/serial/port> info_corefile </path/to/program/elf/file>` or `espcoredump.py -p </path/to/serial/port> dbg_corefile </path/to/program/elf/file>`

5.11.4 Print core dump to UART

When this option is selected base64-encoded core dumps are printed on UART upon system panic. In this case user should save core dump text body to some file manually and then run the following command: `espcoredump.py info_corefile -t b64 -c </path/to/saved/base64/text> </path/to/program/elf/file>` or `espcoredump.py dbg_corefile -t b64 -c </path/to/saved/base64/text> </path/to/program/elf/file>`

Base64-encoded body of core dump will be between the following header and footer:

```
===== CORE DUMP START =====
<body of base64-encoded core dump, save it to file on disk>
===== CORE DUMP END =====
```

The `CORE DUMP START` and `CORE DUMP END` lines must not be included in core dump text file.

5.11.5 ROM Functions in Backtraces

It is possible situation that at the moment of crash some tasks or/and crashed task itself have one or more ROM functions in their callstacks. Since ROM is not part of the program ELF it will be impossible for GDB to parse such callstacks, because it tries to analyse functions' prologues to accomplish that. In that case callstack printing will be broken with error message at the first ROM function. To overcome this issue you can use ROM ELF provided by Espressif (https://dl.espressif.com/dl/esp32_rom.elf) and pass it to 'espcoredump.py'.

5.11.6 Running 'espcoredump.py'

Generic command syntax:

```
espcoredump.py [options] command [args]
```

Script Options

- `-chip,-c {auto,esp32}`. Target chip type. Supported values are `auto` and `esp32`.

- `-port,-p PORT`. Serial port device.
- `-baud,-b BAUD`. Serial port baud rate used when flashing/reading.

Commands

- `info_corefile`. Retrieve core dump and print useful info.
- `dbg_corefile`. Retrieve core dump and start GDB session with it.

Command Arguments

- `-debug,-d DEBUG`. Log level (0..3).
- `-gdb,-g GDB`. Path to gdb to use for data retrieval.
- `-core,-c CORE`. Path to core dump file to use (if skipped core dump will be read from flash).
- `-core-format,-t CORE_FORMAT`. Specifies that file passed with `“-c”` is an ELF (`“elf”`), dumped raw binary (`“raw”`) or base64-encoded (`“b64”`) format.
- `-off,-o OFF`. Offset of coredump partition in flash (type *idf.py partition_table* to see it).
- `-save-core,-s SAVE_CORE`. Save core to file. Otherwise temporary core file will be deleted. Ignored with `“-c”`.
- `-rom-elf,-r ROM_ELF`. Path to ROM ELF file to use (if skipped `“esp32_rom.elf”` is used).
- `-print-mem,-m` Print memory dump. Used only with `“info_corefile”`.
- `<prog>` Path to program ELF file.

5.12 Event Handling

Several ESP-IDF components use *events* to inform application about state changes, such as connection or disconnection. This document gives an overview of these event mechanisms.

5.12.1 Wi-Fi, Ethernet, and IP Events

Before the introduction of *esp_event library*, events from Wi-Fi driver, Ethernet driver, and TCP/IP stack were dispatched using the so-called *legacy event loop*. The following sections explain each of the methods.

esp_event Library Event Loop

esp_event library is designed to supersede the legacy event loop for the purposes of event handling in ESP-IDF. In the legacy event loop, all possible event types and event data structures had to be defined in *system_event_id_t* enumeration and *system_event_info_t* union, which made it impossible to send custom events to the event loop, and use the event loop for other kinds of events (e.g. Mesh). Legacy event loop also supported only one event handler function, therefore application components could not handle some of Wi-Fi or IP events themselves, and required application to forward these events from its event handler function.

See *esp_event library API reference* for general information on using this library. Wi-Fi, Ethernet, and IP events are sent to the *default event loop* provided by this library.

Legacy Event Loop

This event loop implementation is started using *esp_event_loop_init()* function. Application typically supplies an *event handler*, a function with the following signature:

```
esp_err_t event_handler(void *ctx, system_event_t *event)
{
}
```

Both the pointer to event handler function, and an arbitrary context pointer are passed to *esp_event_loop_init()*.

When Wi-Fi, Ethernet, or IP stack generate an event, this event is sent to a high-priority *event* task via a queue. Application-provided event handler function is called in the context of this task. Event task stack

size and event queue size can be adjusted using `CONFIG_ESP_SYSTEM_EVENT_TASK_STACK_SIZE` and `CONFIG_ESP_SYSTEM_EVENT_QUEUE_SIZE` options, respectively.

Event handler receives a pointer to the event structure (`system_event_t`) which describes current event. This structure follows a *tagged union* pattern: `event_id` member indicates the type of event, and `event_info` member is a union of description structures. Application event handler will typically use `switch(event->event_id)` to handle different kinds of events.

If application event handler needs to relay the event to some other task, it is important to note that event pointer passed to the event handler is a pointer to temporary structure. To pass the event to another task, application has to make a copy of the entire structure.

Event IDs and Corresponding Data Structures

| Event ID (legacy event ID) | Event data structure |
|---|---|
| Wi-Fi | |
| WIFI_EVENT_WIFI_READY (SYSTEM_EVENT_WIFI_READY) | n/a |
| WIFI_EVENT_SCAN_DONE (SYSTEM_EVENT_SCAN_DONE) | <code>wifi_event_sta_scan_done_t</code> |
| WIFI_EVENT_STA_START (SYSTEM_EVENT_STA_START) | n/a |
| WIFI_EVENT_STA_STOP (SYSTEM_EVENT_STA_STOP) | n/a |
| WIFI_EVENT_STA_CONNECTED (SYSTEM_EVENT_STA_CONNECTED) | <code>wifi_event_sta_connected_t</code> |
| WIFI_EVENT_STA_DISCONNECTED (SYSTEM_EVENT_STA_DISCONNECTED) | <code>wifi_event_sta_disconnected_t</code> |
| WIFI_EVENT_STA_AUTHMODE_CHANGE (SYSTEM_EVENT_STA_AUTHMODE_CHANGE) | <code>wifi_event_sta_authmode_change_t</code> |
| WIFI_EVENT_STA_WPS_ER_SUCCESS (SYSTEM_EVENT_STA_WPS_ER_SUCCESS) | n/a |
| WIFI_EVENT_STA_WPS_ER_FAILED (SYSTEM_EVENT_STA_WPS_ER_FAILED) | <code>wifi_event_sta_wps_fail_reason_t</code> |
| WIFI_EVENT_STA_WPS_ER_TIMEOUT (SYSTEM_EVENT_STA_WPS_ER_TIMEOUT) | n/a |
| WIFI_EVENT_STA_WPS_ER_PIN (SYSTEM_EVENT_STA_WPS_ER_PIN) | <code>wifi_event_sta_wps_er_pin_t</code> |
| WIFI_EVENT_AP_START (SYSTEM_EVENT_AP_START) | n/a |
| WIFI_EVENT_AP_STOP (SYSTEM_EVENT_AP_STOP) | n/a |
| WIFI_EVENT_AP_STACONNECTED (SYSTEM_EVENT_AP_STACONNECTED) | <code>wifi_event_ap_staconnected_t</code> |
| WIFI_EVENT_AP_STADISCONNECTED (SYSTEM_EVENT_AP_STADISCONNECTED) | <code>wifi_event_ap_stadisconnected_t</code> |
| WIFI_EVENT_AP_PROBEREQRCVD (SYSTEM_EVENT_AP_PROBEREQRCVD) | <code>wifi_event_ap_probe_req_rx_t</code> |
| Ethernet | |
| ETHERNET_EVENT_START (SYSTEM_EVENT_ETH_START) | n/a |
| ETHERNET_EVENT_STOP (SYSTEM_EVENT_ETH_STOP) | n/a |
| ETHERNET_EVENT_CONNECTED (SYSTEM_EVENT_ETH_CONNECTED) | n/a |
| ETHERNET_EVENT_DISCONNECTED (SYSTEM_EVENT_ETH_DISCONNECTED) | n/a |
| IP | |
| IP_EVENT_STA_GOT_IP (SYSTEM_EVENT_STA_GOT_IP) | <code>ip_event_got_ip_t</code> |
| IP_EVENT_STA_LOST_IP (SYSTEM_EVENT_STA_LOST_IP) | n/a |
| IP_EVENT_AP_STAIPASSIGNED (SYSTEM_EVENT_AP_STAIPASSIGNED) | n/a |
| IP_EVENT_GOT_IP6 (SYSTEM_EVENT_GOT_IP6) | <code>ip_event_got_ip6_t</code> |
| IP_EVENT_ETH_GOT_IP (SYSTEM_EVENT_ETH_GOT_IP) | <code>ip_event_got_ip_t</code> |

5.12.2 Mesh Events

ESP-MESH uses a system similar to the *Legacy Event Loop* to deliver events to the application. See *System Events* for details.

5.12.3 Bluetooth Events

Various modules of the Bluetooth stack deliver events to applications via dedicated callback functions. Callback functions receive the event type (enumerated value) and event data (union of structures for each event type). The following list gives the registration API name, event enumeration type, and event parameters type.

- BLE GAP: `esp_ble_gap_register_callback()`, `esp_gap_ble_cb_event_t`, `esp_ble_gap_cb_param_t`.
- BT GAP: `esp_bt_gap_register_callback()`, `esp_bt_gap_cb_event_t`, `esp_bt_gap_cb_param_t`.
- GATT: `esp_ble_gattc_register_callback()`, `esp_bt_gattc_cb_event_t`, `esp_bt_gattc_cb_param_t`.
- GATTS: `esp_ble_gatts_register_callback()`, `esp_bt_gatts_cb_event_t`, `esp_bt_gatts_cb_param_t`.
- SPP: `esp_spp_register_callback()`, `esp_spp_cb_event_t`, `esp_spp_cb_param_t`.
- Blufi: `esp_blufi_register_callbacks()`, `esp_blufi_cb_event_t`, `esp_blufi_cb_param_t`.
- A2DP: `esp_a2d_register_callback()`, `esp_a2d_cb_event_t`, `esp_a2d_cb_param_t`.
- AVRC: `esp_avrc_ct_register_callback()`, `esp_avrc_ct_cb_event_t`, `esp_avrc_ct_cb_param_t`.
- HFP Client: `esp_hf_client_register_callback()`, `esp_hf_client_cb_event_t`, `esp_hf_client_cb_param_t`.
- HFP AG: `esp_hf_ag_register_callback()`, `esp_hf_ag_cb_event_t`, `esp_hf_ag_cb_param_t`.

5.13 Support for external RAM

5.13.1 Introduction

ESP32 has a few hundred kilobytes of internal RAM, residing on the same die as the rest of the chip components. It can be insufficient for some purposes, so ESP32 has the ability to also use up to 4 MB of external SPI RAM memory. The external memory is incorporated in the memory map and, with certain restrictions, is usable in the same way as internal data RAM.

5.13.2 Hardware

ESP32 supports SPI PSRAM connected in parallel with the SPI flash chip. While ESP32 is capable of supporting several types of RAM chips, the ESP32 SDK only supports the ESP-PSRAM32 chip at the moment.

The ESP-PSRAM32 chip is a 1.8 V device which can only be used in parallel with a 1.8 V flash component. Make sure to either set the MTDI pin to a high signal level on bootup, or program ESP32 eFuses to always use the VDD_SIO level of 1.8 V. Not doing this can damage the PSRAM and/or flash chip.

To connect the ESP-PSRAM32 chip to ESP32D0W*, connect the following signals:

- PSRAM /CE (pin 1) > ESP32 GPIO 16
- PSRAM SO (pin 2) > flash DO
- PSRAM SIO[2] (pin 3) > flash WP
- PSRAM SI (pin 5) > flash DI
- PSRAM SCLK (pin 6) > ESP32 GPIO 17
- PSRAM SIO[3] (pin 7) > flash HOLD

- PSRAM Vcc (pin 8) > ESP32 VCC_SDIO

Connections for ESP32D2W* chips are TBD.

Note: Espressif produces the line of ESP32-WROVER modules which contain ESP32, 1.8 V flash, and ESP-PSRAM32. These modules are ready to be mounted on an end product PCB.

5.13.3 Configuring External RAM

ESP-IDF fully supports the use of external memory in applications. Once the external RAM is initialized at startup, ESP-IDF can be configured to handle it in several ways:

- *Integrate RAM into the ESP32 memory map*
- *Add external RAM to the capability allocator*
- *Provide external RAM via malloc() (default)*
- *Allow .bss segment placed in external memory*

Integrate RAM into the ESP32 memory map

Select this option by choosing “Integrate RAM into ESP32 memory map” from [CONFIG_SPIRAM_USE](#).

This is the most basic option for external SPI RAM integration. Most likely, you will need another, more advanced option.

During the ESP-IDF startup, external RAM is mapped into the data address space, starting at address 0x3F800000 (byte-accessible). The length of this region is the same as the SPI RAM size (up to the limit of 4 MB).

Applications can manually place data in external memory by creating pointers to this region. So if an application uses external memory, it is responsible for all management of the external SPI RAM: coordinating buffer usage, preventing corruption, etc.

Add external RAM to the capability allocator

Select this option by choosing “Make RAM allocatable using heap_caps_malloc(..., MALLOC_CAP_SPIRAM)” from [CONFIG_SPIRAM_USE](#).

When enabled, memory is mapped to address 0x3F800000 and also added to the *capabilities-based heap memory allocator* using `MALLOC_CAP_SPIRAM`.

To allocate memory from external RAM, a program should call `heap_caps_malloc(size, MALLOC_CAP_SPIRAM)`. After use, this memory can be freed by calling the normal `free()` function.

Provide external RAM via malloc()

Select this option by choosing “Make RAM allocatable using malloc() as well” from [CONFIG_SPIRAM_USE](#). This is the default option.

In this case, memory is added to the capability allocator as described for the previous option. However, it is also added to the pool of RAM that can be returned by the standard `malloc()` function.

This allows any application to use the external RAM without having to rewrite the code to use `heap_caps_malloc(..., MALLOC_CAP_SPIRAM)`.

An additional configuration item, [CONFIG_SPIRAM_MALLOC_ALWAYSINTERNAL](#), can be used to set the size threshold when a single allocation should prefer external memory:

- When allocating a size less than the threshold, the allocator will try internal memory first.
- When allocating a size equal to or larger than the threshold, the allocator will try external memory first.

If a suitable block of preferred internal/external memory is not available, the allocator will try the other type of memory.

Because some buffers can only be allocated in internal memory, a second configuration item `CONFIG_SPIRAM_MALLOC_RESERVE_INTERNAL` defines a pool of internal memory which is reserved for *only* explicitly internal allocations (such as memory for DMA use). Regular `malloc()` will not allocate from this pool. The `MALLOC_CAP_DMA` and `MALLOC_CAP_INTERNAL` flags can be used to allocate memory from this pool.

Allow .bss segment placed in external memory

Enable this option by checking `CONFIG_SPIRAM_ALLOW_BSS_SEG_EXTERNAL_MEMORY`. This configuration setting is independent of the other three.

If enabled, a region of the address space starting from 0x3F800000 will be used to store zero-initialized data (BSS segment) from the lwIP, net80211, libpp, and bluebird ESP-IDF libraries.

Additional data can be moved from the internal BSS segment to external RAM by applying the macro `EXT_RAM_ATTR` to any static declaration (which is not initialized to a non-zero value).

This option reduces the internal static memory used by the BSS segment.

Remaining external RAM can also be added to the capability heap allocator using the method shown above.

5.13.4 Restrictions

External RAM use has the following restrictions:

- When flash cache is disabled (for example, if the flash is being written to), the external RAM also becomes inaccessible; any reads from or writes to it will lead to an illegal cache access exception. This is also the reason why ESP-IDF does not by default allocate any task stacks in external RAM (see below).
- External RAM cannot be used as a place to store DMA transaction descriptors or as a buffer for a DMA transfer to read from or write into. Any buffers that will be used in combination with DMA must be allocated using `heap_caps_malloc(size, MALLOC_CAP_DMA)` and can be freed using a standard `free()` call.
- External RAM uses the same cache region as the external flash. This means that frequently accessed variables in external RAM can be read and modified almost as quickly as in internal ram. However, when accessing large chunks of data (>32 KB), the cache can be insufficient, and speeds will fall back to the access speed of the external RAM. Moreover, accessing large chunks of data can “push out” cached flash, possibly making the execution of code slower afterwards.
- External RAM cannot be used as task stack memory. Due to this, `xTaskCreate()` and similar functions will always allocate internal memory for stack and task TCBS, and functions such as `xTaskCreateStatic()` will check if the buffers passed are internal. However, for tasks not calling on code in ROM in any way, directly or indirectly, the menuconfig option `CONFIG_SPIRAM_ALLOW_STACK_EXTERNAL_MEMORY` will eliminate the check in `xTaskCreateStatic`, allowing a task’s stack to be in external RAM. Using this is not advised, however.
- By default, failure to initialize external RAM will cause the ESP-IDF startup to abort. This can be disabled by enabling the config item `CONFIG_SPIRAM_IGNORE_NOTFOUND`. If `CONFIG_SPIRAM_ALLOW_BSS_SEG_EXTERNAL_MEMORY` is enabled, the option to ignore failure is not available as the linker will have assigned symbols to external memory addresses at link time.
- When used at 80 MHz clock speed, external RAM must also occupy either the HSPI or VSPI bus. Select which SPI host will be used by `CONFIG_SPIRAM_OCCUPY_SPI_HOST`.

5.13.5 Chip revisions

There are some issues with certain revisions of ESP32 that have repercussions for use with external RAM. The issues are documented in the ESP32 [ECO](#) document. In particular, ESP-IDF handles the bugs mentioned in the following ways:

ESP32 rev v0

ESP-IDF has no workaround for the bugs in this revision of silicon, and it cannot be used to map external PSRAM into ESP32's main memory map.

ESP32 rev v1

The bugs in this revision of silicon cause issues if certain sequences of machine instructions operate on external memory. (ESP32 ECO 3.2). As a workaround, the GCC compiler received the flag `-mfix-esp32-psram-cache-issue` to filter these sequences and only output the code that can safely be executed. Enable this flag by checking `CONFIG_SPIRAM_CACHE_WORKAROUND`.

Aside from linking to a recompiled version of Newlib with the additional flag, ESP-IDF also does the following:

- Avoids using some ROM functions
- Allocates static memory for the WiFi stack

5.14 Fatal Errors

5.14.1 Overview

In certain situations, execution of the program can not be continued in a well defined way. In ESP-IDF, these situations include:

- CPU Exceptions: Illegal Instruction, Load/Store Alignment Error, Load/Store Prohibited error, Double Exception.
- System level checks and safeguards:
 - *Interrupt watchdog* timeout
 - *Task watchdog* timeout (only fatal if `CONFIG_ESP_TASK_WDT_PANIC` is set)
 - Cache access error
 - Brownout detection event
 - Stack overflow
 - Stack smashing protection check
 - Heap integrity check
- Failed assertions, via `assert`, `configASSERT` and similar macros.

This guide explains the procedure used in ESP-IDF for handling these errors, and provides suggestions on troubleshooting the errors.

5.14.2 Panic Handler

Every error cause listed in the *Overview* will be handled by *panic handler*.

Panic handler will start by printing the cause of the error to the console. For CPU exceptions, the message will be similar to:

```
Guru Meditation Error: Core 0 panic'ed (IllegalInstruction). Exception was
↳unhandled.
```

For some of the system level checks (interrupt watchdog, cache access error), the message will be similar to:

```
Guru Meditation Error: Core 0 panic'ed (Cache disabled but cached memory region
↳accessed)
```

In all cases, error cause will be printed in parens. See *Guru Meditation Errors* for a list of possible error causes.

Subsequent behavior of the panic handler can be set using `CONFIG_ESP32_PANIC` configuration choice. The available options are:

- Print registers and reboot (`CONFIG_ESP32_PANIC_PRINT_REBOOT`) —default option.
This will print register values at the point of the exception, print the backtrace, and restart the chip.
- Print registers and halt (`CONFIG_ESP32_PANIC_PRINT_HALT`)
Similar to the above option, but halt instead of rebooting. External reset is required to restart the program.
- Silent reboot (`CONFIG_ESP32_PANIC_SILENT_REBOOT`)
Don't print registers or backtrace, restart the chip immediately.
- Invoke GDB Stub (`CONFIG_ESP32_PANIC_GDBSTUB`)
Start GDB server which can communicate with GDB over console UART port. See *GDB Stub* for more details.

Behavior of panic handler is affected by two other configuration options.

- If `CONFIG_ESP32_DEBUG_OCDAWARE` is enabled (which is the default), panic handler will detect whether a JTAG debugger is connected. If it is, execution will be halted and control will be passed to the debugger. In this case registers and backtrace are not dumped to the console, and GDBStub / Core Dump functions are not used.
- If *Core Dump* feature is enabled (`CONFIG_ESP32_ENABLE_COREDUMP_TO_FLASH` or `CONFIG_ESP32_ENABLE_COREDUMP_TO_UART` options), then system state (task stacks and registers) will be dumped either to Flash or UART, for later analysis.
- If `CONFIG_ESP_PANIC_HANDLER_IRAM` is disabled (disabled by default), the panic handler code is placed in flash memory not IRAM. This means that if ESP-IDF crashes while flash cache is disabled, the panic handler will automatically re-enable flash cache before running GDB Stub or Core Dump. This adds some minor risk, if the flash cache status is also corrupted during the crash.
If this option is enabled, the panic handler code is placed in IRAM. This allows the panic handler to run without needing to re-enable cache first. This may be necessary to debug some complex issues with crashes while flash cache is disabled (for example, when writing to SPI flash).

The following diagram illustrates panic handler behavior:

5.14.3 Register Dump and Backtrace

Unless `CONFIG_ESP32_PANIC_SILENT_REBOOT` option is enabled, panic handler prints some of the CPU registers, and the backtrace, to the console:

```
Core 0 register dump:
PC      : 0x400e14ed  PS      : 0x00060030  A0      : 0x800d0805  A1      : _
↳0x3ffb5030
A2      : 0x00000000  A3      : 0x00000001  A4      : 0x00000001  A5      : _
↳0x3ffb50dc
A6      : 0x00000000  A7      : 0x00000001  A8      : 0x00000000  A9      : _
↳0x3ffb5000
A10     : 0x00000000  A11     : 0x3ffb2bac  A12     : 0x40082d1c  A13     : _
↳0x06ff1ff8
A14     : 0x3ffb7078  A15     : 0x00000000  SAR     : 0x00000014  EXCCAUSE: _
↳0x0000001d
EXCVADDR: 0x00000000  LBEG    : 0x4000c46c  LEND    : 0x4000c477  LCOUNT : _
↳0xffffffff

Backtrace: 0x400e14ed:0x3ffb5030 0x400d0802:0x3ffb5050
```

Register values printed are the register values in the exception frame, i.e. values at the moment when CPU exception or other fatal error has occurred.

Register dump is not printed if the panic handler was executed as a result of an `abort()` call.

In some cases, such as interrupt watchdog timeout, panic handler may print additional CPU registers (EPC1-EPC4) and the registers/backtrace of the code running on the other CPU.

Backtrace line contains PC:SP pairs, where PC is the Program Counter and SP is Stack Pointer, for each stack frame of the current task. If a fatal error happens inside an ISR, the backtrace may include PC:SP pairs both from the task which was interrupted, and from the ISR.

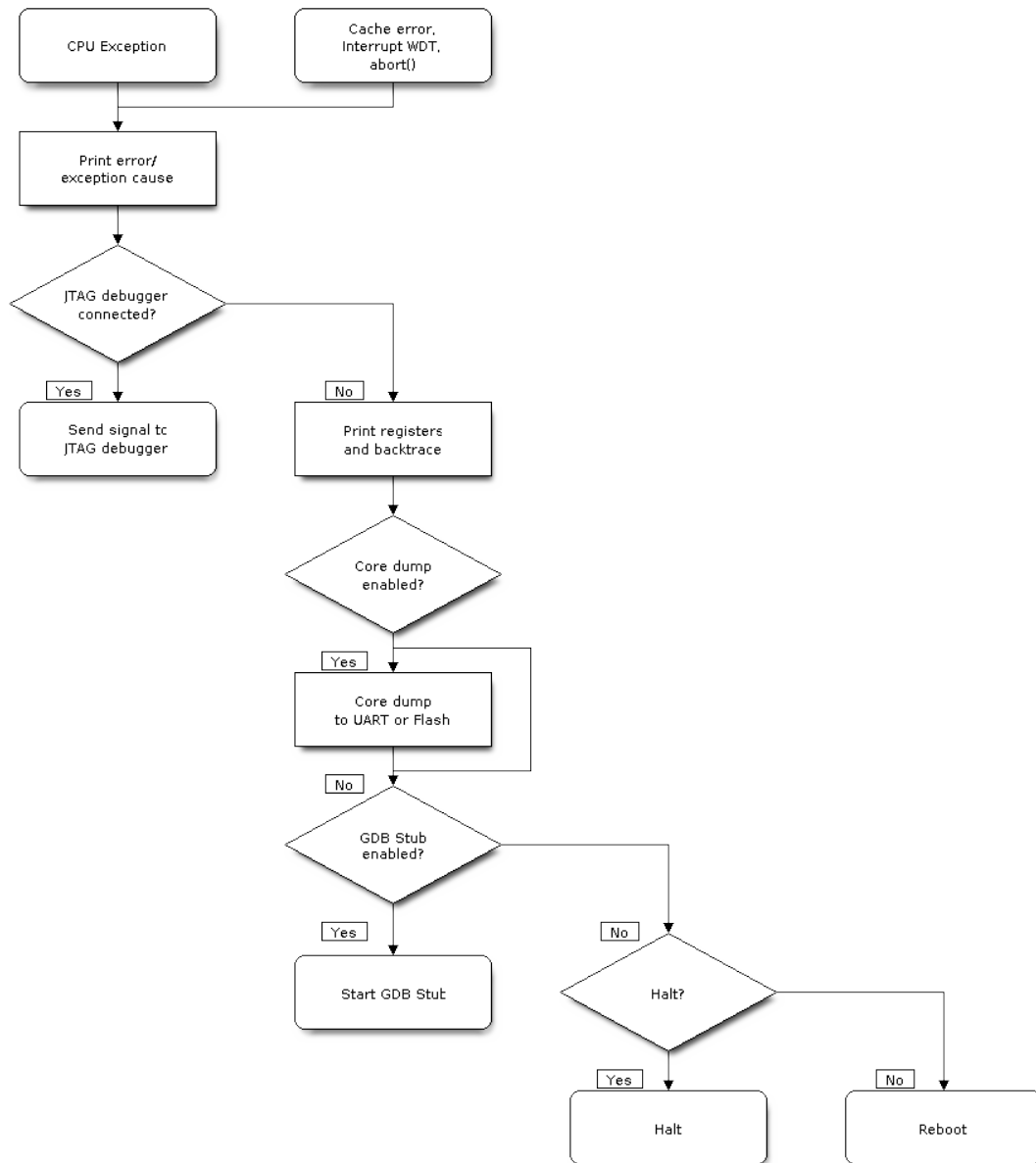


Fig. 32: Panic Handler Flowchart (click to enlarge)

If *IDF Monitor* is used, Program Counter values will be converted to code locations (function name, file name, and line number), and the output will be annotated with additional lines:

```
Core 0 register dump:
PC      : 0x400e14ed PS      : 0x00060030 A0      : 0x800d0805 A1      : _
↳0x3ffb5030
0x400e14ed: app_main at /Users/user/esp/example/main/main.cpp:36

A2      : 0x00000000 A3      : 0x00000001 A4      : 0x00000001 A5      : _
↳0x3ffb50dc
A6      : 0x00000000 A7      : 0x00000001 A8      : 0x00000000 A9      : _
↳0x3ffb5000
A10     : 0x00000000 A11     : 0x3ffb2bac A12     : 0x40082d1c A13     : _
↳0x06ff1fff
0x40082d1c: _calloc_r at /Users/user/esp/esp-idf/components/newlib/syscalls.c:51

A14     : 0x3ffb7078 A15     : 0x00000000 SAR      : 0x00000014 EXCCAUSE:_
↳0x0000001d
EXCVADDR: 0x00000000 LBEG    : 0x4000c46c LEND     : 0x4000c477 LCOUNT  : _
↳0xffffffff

Backtrace: 0x400e14ed:0x3ffb5030 0x400d0802:0x3ffb5050
0x400e14ed: app_main at /Users/user/esp/example/main/main.cpp:36

0x400d0802: main_task at /Users/user/esp/esp-idf/components/esp32/cpu_start.c:470
```

To find the location where a fatal error has happened, look at the lines which follow the “Backtrace” line. Fatal error location is the top line, and subsequent lines show the call stack.

5.14.4 GDB Stub

If `CONFIG_ESP32_PANIC_GDBSTUB` option is enabled, panic handler will not reset the chip when fatal error happens. Instead, it will start GDB remote protocol server, commonly referred to as GDB Stub. When this happens, GDB instance running on the host computer can be instructed to connect to the ESP32 UART port.

If *IDF Monitor* is used, GDB is started automatically when GDB Stub prompt is detected on the UART. The output would look like this:

```
Entering gdb stub now.
$T0b#e6GNU gdb (crosstool-NG crosstool-ng-1.22.0-80-gff1f415) 7.10
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=x86_64-build_apple-darwin16.3.0 --target=xtensa-
↳esp32-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from /Users/user/esp/example/build/example.elf...done.
Remote debugging using /dev/cu.usbserial-31301
0x400e1b41 in app_main ()
    at /Users/user/esp/example/main/main.cpp:36
36      *((int*) 0) = 0;
(gdb)
```

GDB prompt can be used to inspect CPU registers, local and static variables, and arbitrary locations in memory. It

is not possible to set breakpoints, change PC, or continue execution. To reset the program, exit GDB and perform external reset: Ctrl-T Ctrl-R in IDF Monitor, or using external reset button on the development board.

5.14.5 Guru Meditation Errors

This section explains the meaning of different error causes, printed in parens after `Guru Meditation Error: Core panic'ed` message.

Note: See [Wikipedia article](#) for historical origins of “Guru Meditation” .

IllegalInstruction

This CPU exception indicates that the instruction which was executed was not a valid instruction. Most common reasons for this error include:

- FreeRTOS task function has returned. In FreeRTOS, if task function needs to terminate, it should call `vTaskDelete()` function and delete itself, instead of returning.
- Failure to load next instruction from SPI flash. This usually happens if:
 - Application has reconfigured SPI flash pins as some other function (GPIO, UART, etc.). Consult Hardware Design Guidelines and the Datasheet for the chip or module for details about SPI flash pins.
 - Some external device was accidentally connected to SPI flash pins, and has interfered with communication between ESP32 and SPI flash.

InstrFetchProhibited

This CPU exception indicates that CPU could not load an instruction because the the address of the instruction did not belong to a valid region in instruction RAM or ROM.

Usually this means an attempt to call a function pointer, which does not point to valid code. PC (Program Counter) register can be used as an indicator: it will be zero or will contain garbage value (not `0x4xxxxxxxx`).

LoadProhibited, StoreProhibited

This CPU exception happens when application attempts to read from or write to an invalid memory location. The address which was written/read is found in EXCVADDR register in the register dump. If this address is zero, it usually means that application attempted to dereference a NULL pointer. If this address is close to zero, it usually means that application attempted to access member of a structure, but the pointer to the structure was NULL. If this address is something else (garbage value, not in `0x3fxxxxxxxx - 0x6xxxxxxxx` range), it likely means that the pointer used to access the data was either not initialized or was corrupted.

IntegerDivideByZero

Application has attempted to do integer division by zero.

LoadStoreAlignment

Application has attempted to read or write memory location, and address alignment did not match load/store size. For example, 32-bit load can only be done from 4-byte aligned address, and 16-bit load can only be done from a 2-byte aligned address.

LoadStoreError

Application has attempted to do a 8- or 16- bit load/store from a memory region which only supports 32-bit loads/stores. For example, dereferencing a `char*` pointer which points into instruction memory will result in such an error.

Unhandled debug exception

This will usually be followed by a message like:

```
Debug exception reason: Stack canary watchpoint triggered (task_name)
```

This error indicates that application has written past the end of the stack of `task_name` task. Note that not every stack overflow is guaranteed to trigger this error. It is possible that the task writes to stack beyond the stack canary location, in which case the watchpoint will not be triggered.

Interrupt wdt timeout on CPU0 / CPU1

Indicates that interrupt watchdog timeout has occurred. See [Watchdogs](#) for more information.

Cache disabled but cached memory region accessed

In some situations ESP-IDF will temporarily disable access to external SPI Flash and SPI RAM via caches. For example, this happens with `spi_flash` APIs are used to read/write/erase/mmap regions of SPI Flash. In these situations, tasks are suspended, and interrupt handlers not registered with `ESP_INTR_FLAG_IRAM` are disabled. Make sure that any interrupt handlers registered with this flag have all the code and data in IRAM/DRAM. Refer to the [SPI flash API documentation](#) for more details.

5.14.6 Other Fatal Errors

Brownout

ESP32 has a built-in brownout detector, which is enabled by default. Brownout detector can trigger system reset if supply voltage goes below safe level. Brownout detector can be configured using `CONFIG_ESP32_BROWNOUT_DET` and `CONFIG_ESP32_BROWNOUT_DET_LVL_SEL` options. When brownout detector triggers, the following message is printed:

```
Brownout detector was triggered
```

Chip is reset after the message is printed.

Note that if supply voltage is dropping at a fast rate, only part of the message may be seen on the console.

Corrupt Heap

ESP-IDF heap implementation contains a number of run-time checks of heap structure. Additional checks (“Heap Poisoning”) can be enabled in menuconfig. If one of the checks fails, message similar to the following will be printed:

```
CORRUPT HEAP: Bad tail at 0x3ffe270a. Expected 0xbaad5678 got 0xbaac5678
assertion "head != NULL" failed: file "/Users/user/esp/esp-idf/components/heap/
↳multi_heap_poisoning.c", line 201, function: multi_heap_free
abort() was called at PC 0x400dca43 on core 0
```

Consult [Heap Memory Debugging](#) documentation for further information.

Stack Smashing

Stack smashing protection (based on GCC `-fstack-protector*` flags) can be enabled in ESP-IDF using `CONFIG_COMPILER_STACK_CHECK_MODE` option. If stack smashing is detected, message similar to the following will be printed:

```
Stack smashing protect failure!

abort() was called at PC 0x400d2138 on core 0

Backtrace: 0x4008e6c0:0x3ffc1780 0x4008e8b7:0x3ffc17a0 0x400d2138:0x3ffc17c0
↳0x400e79d5:0x3ffc17e0 0x400e79a7:0x3ffc1840 0x400e79df:0x3ffc18a0
↳0x400e2235:0x3ffc18c0 0x400e1916:0x3ffc18f0 0x400e19cd:0x3ffc1910
↳0x400e1a11:0x3ffc1930 0x400e1bb2:0x3ffc1950 0x400d2c44:0x3ffc1a80
0
```

The backtrace should point to the function where stack smashing has occurred. Check the function code for unbounded access to local arrays.

5.15 Flash Encryption

This document provides introduction to Flash encryption concept on ESP32 and demonstrates how this feature can be used during development as well as production by the user using a sample example. The primary intention of the document is to act as a quick start guide to test and verify flash encryption operations. The details of the flash encryption block can be found in the [ESP32 Technical reference manual](#).

5.15.1 Introduction

Flash encryption is a feature for encrypting the contents of the ESP32's attached SPI flash. When flash encryption is enabled, physical readout of the SPI flash is not sufficient to recover most flash contents. Encryption is applied by flashing the ESP32 with plaintext data, and (if encryption is enabled) the bootloader encrypts the data in place on first boot.

With flash encryption enabled, following kinds of flash data are encrypted by default:

- Bootloader
- Partition Table
- All “app” type partitions

Other types of flash data are encrypted conditionally:

- Secure boot bootloader digest (if secure boot is enabled)
- Any partition marked with the “encrypted” flag in the partition table

Flash encryption is separate from the *Secure Boot* feature, and you can use flash encryption without enabling secure boot. However, for a secure environment both should be used simultaneously.

Important: For production use, flash encryption should be enabled in the “Release” mode only.

Important: Enabling flash encryption limits the options for further updates of the ESP32. Make sure to read this document (including *Limitations of Flash Encryption*) and understand the implications of enabling flash encryption.

5.15.2 eFuse Used During Flash Encryption Process

The flash encryption operation is controlled by various eFuses available on ESP32. Below is the list of eFuse and their description:

| eFuse | Description | Can be |
|----------------------|--|----------|
| locked for writing | Default Value | reading/ |
| Coding scheme | This 2 bit wide eFuse controls the actual number of bits to be used from BLOCK1 to derive final 256 bit AES key. The coding scheme value is decoded as below: 0: 256 bits 1: 192 bits 2: 128 bits Final AES key is derived based on the FLASH_CRYPT_CONFIG value | Yes |
| BLOCK1 | 256 bit wide eFuse block for storing AES key | Yes |
| FLASH_CRYPT_CONFIG | 4 bit wide eFuse which controls the AES encryption process | Yes |
| download_dis_encrypt | When set, disables the flash encryption operation while running in UART download mode | Yes |
| download_dis_decrypt | When set, disables the flash decryption operation while running in UART download mode | Yes |
| FLASH_CRYPT_CNT | 7 bit eFuse which enables/disables encryption at boot time Even number of bits set (0, 2, 4, 6): encrypt flash at boot time Odd number of bits set (1, 3, 5, 7): do not encrypt flash at boot time | Yes |

Read and write access to above bits is controlled by appropriate bits in `efuse_wr_disable` and `efuse_rd_disable` registers. More information about ESP32 eFuse can be found at [eFuse manager](#).

5.15.3 Flash Encryption Process

Assuming the eFuse values are in default state and second stage bootloader is compiled to support flash encryption, the flash encryption process executes as below:

- On first power-on reset, all data in flash is un-encrypted (plaintext). First stage loader (ROM) will load the second stage loader in IRAM.
- Second stage bootloader will read the `flash_crypt_cnt` (=00000000b) eFuse value and since the value is 0 (even number of bits set) it will configure and enable the flash encryption block. It will also program

FLASH_CRYPT_CFG eFuse to value 0xF.

- The flash encryption block will generate AES-256 bit key and store into BLOCK1 eFuse. This operation is performed in hardware and the key can not be accessed by software.
- Next the flash encryption block will encrypt the flash contents (based on partition table flag value). Encrypting in-place can take some time (up to a minute for large partitions).
- Second stage bootloader then sets the first available bit in flash_crypt_cnt (=00000001b) to mark the flash contents as encrypted (odd number of bits set).
- For *Release Mode* second stage bootloader will program download_dis_encrypt, download_dis_decrypt & download_dis_cache eFuse bits to 1 to prevent UART bootloader from decrypting the flash contents. It will also write protect the FLASH_CRYPT_CNT eFuse bits.
- For *Development Mode* second stage bootloader will program only download_dis_decrypt & download_dis_cache eFuse bits to allow UART bootloader reflashing of encrypted binaries. Also FLASH_CRYPT_CNT eFuse bits will NOT be write protected.
- The second stage bootloader then reboots the device to start executing encrypted image. It will transparently decrypt the flash contents and load into IRAM.

During development stage there is a frequent need to program different plaintext flash images and test the flash encryption process. This requires UART download mode to be able to load new plaintext images as many number of times as required. However during manufacturing or production UART download mode should not be allowed to access flash contents due to security reason. Hence this requires two different ESP32 configurations: one for development and other for production. Following section describes *Development Mode* and *Release Mode* for flash encryption and a step by step process to use them.

Important: Development mode as the name suggests should be used ONLY DURING DEVELOPMENT as it does not prevent modification and possible read back of encrypted flash contents.

5.15.4 Steps to Setup Flash Encryption

Development Mode

It is possible to run flash encryption process for development using either ESP32 internally generated key or external host generated keys.

Using ESP32 Generated Flash Encryption Key

As mentioned above *Development Mode* allows user to download as many plaintext images using UART download mode. Following steps needs to be done to test flash encryption process:

- Ensure you have a ESP32 device with default flash encryption eFuse settings as shown in *eFuse Used During Flash Encryption Process*.
- Navigate to flash encryption sample application in \$IDF_PATH/examples/security/flash_encryption folder. This sample application will print the status of flash encryption: enabled or disabled. It will print the FLASH_CRYPT_CNT eFuse value.
- Enable flash encryption support in second stage bootloader. In *Project Configuration Menu*, navigate to “Security Features” .
- Select *Enable flash encryption on boot*.
- By default the mode is set for **Development**.
- Select appropriate Bootloader log verbosity under Bootloader config.
- Update to the partition table offset may be required since after enabling flash encryption the size of bootloader is increased. See *Bootloader Size*
- Save the configuration and exit.

Build and flash the complete image including: bootloader, partition table and app. These partitions are initially written to the flash unencrypted.

```
idf.py flash monitor
```

Once the flashing is complete device will reset and on next boot second stage bootloader will encrypt the flash app partition and then reset. Now the sample application would get decrypted at runtime and executed. Below is a sample output when ESP32 boots after flash encryption is enabled for the first time.

```
--- idf_monitor on /dev/cu.SLAB_USBtoUART 115200 ---
--- Quit: Ctrl+] | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H ---
ets Jun  8 2016 00:22:57

rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0018,len:4
load:0x3fff001c,len:8452
load:0x40078000,len:13608
load:0x40080400,len:6664
entry 0x40080764
I (28) boot: ESP-IDF v4.0-dev-850-gc4447462d-dirty 2nd stage bootloader
I (29) boot: compile time 15:37:14
I (30) boot: Enabling RNG early entropy source...
I (35) boot: SPI Speed      : 40MHz
I (39) boot: SPI Mode      : DIO
I (43) boot: SPI Flash Size : 4MB
I (47) boot: Partition Table:
I (51) boot: ## Label                Usage            Type ST Offset   Length
I (58) boot:  0 nvs                   WiFi data        01 02 0000a000 00006000
I (66) boot:  1 phy_init                RF data          01 01 00010000 00001000
I (73) boot:  2 factory                 factory app      00 00 00020000 00100000
I (81) boot: End of partition table
I (85) esp_image: segment 0: paddr=0x00020020 vaddr=0x3f400020
↳size=0x0808c ( 32908) map
I (105) esp_image: segment 1: paddr=0x000280b4 vaddr=0x3ffb0000
↳size=0x01ea4 ( 7844) load
I (109) esp_image: segment 2: paddr=0x00029f60 vaddr=0x40080000
↳size=0x00400 ( 1024) load
0x40080000: _WindowOverflow4 at esp-idf/esp-idf/components/freertos/
↳xtensa_vectors.S:1778

I (114) esp_image: segment 3: paddr=0x0002a368 vaddr=0x40080400
↳size=0x05ca8 ( 23720) load
I (132) esp_image: segment 4: paddr=0x00030018 vaddr=0x400d0018
↳size=0x126a8 ( 75432) map
0x400d0018: _flash_cache_start at ???

I (159) esp_image: segment 5: paddr=0x000426c8 vaddr=0x400860a8
↳size=0x01f4c ( 8012) load
0x400860a8: prvAddNewTaskToReadyList at esp-idf/esp-idf/components/
↳freertos/tasks.c:4561

I (168) boot: Loaded app from partition at offset 0x20000
I (168) boot: Checking flash encryption...
I (168) flash_encrypt: Generating new flash encryption key...
I (187) flash_encrypt: Read & write protecting new key...
I (187) flash_encrypt: Setting CRYPT_CONFIG efuse to 0xF
W (188) flash_encrypt: Not disabling UART bootloader encryption
I (195) flash_encrypt: Disable UART bootloader decryption...
I (201) flash_encrypt: Disable UART bootloader MMU cache...
I (208) flash_encrypt: Disable JTAG...
I (212) flash_encrypt: Disable ROM BASIC interpreter fallback...
I (219) esp_image: segment 0: paddr=0x00001020 vaddr=0x3fff0018
↳size=0x00004 (    4) (continues on next page)
```


(continued from previous page)

```

I (227) esp_image: segment 1: paddr=0x0000102c vaddr=0x3fff001c
↳size=0x02104 ( 8452)
I (239) esp_image: segment 2: paddr=0x00003138 vaddr=0x40078000
↳size=0x03528 ( 13608)
I (249) esp_image: segment 3: paddr=0x00006668 vaddr=0x40080400
↳size=0x01a08 ( 6664)
I (657) esp_image: segment 0: paddr=0x00020020 vaddr=0x3f400020
↳size=0x0808c ( 32908) map
I (669) esp_image: segment 1: paddr=0x000280b4 vaddr=0x3ffb0000
↳size=0x01ea4 ( 7844)
I (672) esp_image: segment 2: paddr=0x00029f60 vaddr=0x40080000
↳size=0x00400 ( 1024)
0x40080000: _WindowOverflow4 at esp-idf/esp-idf/components/freertos/
↳xtensa_vectors.S:1778

I (676) esp_image: segment 3: paddr=0x0002a368 vaddr=0x40080400
↳size=0x05ca8 ( 23720)
I (692) esp_image: segment 4: paddr=0x00030018 vaddr=0x400d0018
↳size=0x126a8 ( 75432) map
0x400d0018: _flash_cache_start at ???

I (719) esp_image: segment 5: paddr=0x000426c8 vaddr=0x400860a8
↳size=0x01f4c ( 8012)
0x400860a8: prvAddNewTaskToReadyList at esp-idf/esp-idf/components/
↳freertos/tasks.c:4561

I (722) flash_encrypt: Encrypting partition 2 at offset 0x20000...
I (13229) flash_encrypt: Flash encryption completed
I (13229) boot: Resetting with flash encryption enabled...

```

Once the flash encryption is enabled, on subsequent boot the output would mention that flash encryption is already enabled.

```

rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0018,len:4
load:0x3fff001c,len:8452
load:0x40078000,len:13652
ho 0 tail 12 room 4
load:0x40080400,len:6664
entry 0x40080764
I (30) boot: ESP-IDF v4.0-dev-850-gc4447462d-dirty 2nd stage bootloader
I (30) boot: compile time 16:32:53
I (31) boot: Enabling RNG early entropy source...
I (37) boot: SPI Speed      : 40MHz
I (41) boot: SPI Mode      : DIO
I (45) boot: SPI Flash Size : 4MB
I (49) boot: Partition Table:
I (52) boot: ## Label           Usage          Type ST Offset   Length
I (60) boot:  0 nvs             WiFi data     01 02 0000a000 00006000
I (67) boot:  1 phy_init         RF data      01 01 00010000 00001000
I (75) boot:  2 factory          factory app   00 00 00020000 00100000
I (82) boot: End of partition table
I (86) esp_image: segment 0: paddr=0x00020020 vaddr=0x3f400020
↳size=0x0808c ( 32908) map
I (107) esp_image: segment 1: paddr=0x000280b4 vaddr=0x3ffb0000
↳size=0x01ea4 ( 7844) load
I (111) esp_image: segment 2: paddr=0x00029f60 vaddr=0x40080000
↳size=0x00400 ( 1024) load

```

(continues on next page)

(continued from previous page)

```

0x40080000: _WindowOverflow4 at esp-idf/esp-idf/components/freertos/
↳xtensa_vectors.S:1778

I (116) esp_image: segment 3: paddr=0x0002a368 vaddr=0x40080400_
↳size=0x05ca8 ( 23720) load
I (134) esp_image: segment 4: paddr=0x00030018 vaddr=0x400d0018_
↳size=0x126a8 ( 75432) map
0x400d0018: _flash_cache_start at ???

I (162) esp_image: segment 5: paddr=0x000426c8 vaddr=0x400860a8_
↳size=0x01f4c ( 8012) load
0x400860a8: prvAddNewTaskToReadyList at esp-idf/esp-idf/components/
↳freertos/tasks.c:4561

I (171) boot: Loaded app from partition at offset 0x20000
I (171) boot: Checking flash encryption...
I (171) flash_encrypt: flash encryption is enabled (3 plaintext flashes_
↳left)
I (178) boot: Disabling RNG early entropy source...
I (184) cpu_start: Pro cpu up.
I (188) cpu_start: Application information:
I (193) cpu_start: Project name:      flash-encryption
I (198) cpu_start: App version:      v4.0-dev-850-gc4447462d-dirty
I (205) cpu_start: Compile time:     Jun 17 2019 16:32:52
I (211) cpu_start: ELF file SHA256:  8770c886bdf561a7...
I (217) cpu_start: ESP-IDF:          v4.0-dev-850-gc4447462d-dirty
I (224) cpu_start: Starting app cpu, entry point is 0x40080e4c
0x40080e4c: call_start_cpu1 at esp-idf/esp-idf/components/esp32/cpu_start.
↳c:265

I (0) cpu_start: App cpu up.
I (235) heap_init: Initializing. RAM available for dynamic allocation:
I (241) heap_init: At 3FFAE6E0 len 00001920 (6 KiB): DRAM
I (247) heap_init: At 3FFB2EC8 len 0002D138 (180 KiB): DRAM
I (254) heap_init: At 3FFE0440 len 00003AE0 (14 KiB): D/IRAM
I (260) heap_init: At 3FFE4350 len 0001BCB0 (111 KiB): D/IRAM
I (266) heap_init: At 40087FF4 len 0001800C (96 KiB): IRAM
I (273) cpu_start: Pro cpu start user code
I (291) cpu_start: Starting scheduler on PRO CPU.
I (0) cpu_start: Starting scheduler on APP CPU.

Sample program to check Flash Encryption
This is ESP32 chip with 2 CPU cores, WiFi/BT/BLE, silicon revision 1, 4MB_
↳external flash
Flash encryption feature is enabled
Flash encryption mode is DEVELOPMENT
Flash in encrypted mode with flash_crypt_cnt = 1
Halting...

```

At this stage if user wants to update modified plaintext application image to flash in encrypted format it can be done using following command:

```
idf.py encrypted-app-flash monitor
```

Encrypt Multiple Partitions

If all partitions needs to be updated in encrypted format, it can be done as

```
idf.py encrypted-flash monitor
```

Using Host Generated Flash Encryption Key

It is possible to pregenerate the flash encryption key on the host computer and burn it into the ESP32's eFuse key block. This allows data to be pre-encrypted on the host and flashed to the ESP32 without needing a plaintext flash update. This feature allows encrypted flashing in both *Development Mode* and *Release Mode* modes.

Note: This option is not recommended for production unless a separate key is generated for each individual device.

- Ensure you have a ESP32 device with default flash encryption eFuse settings as shown in *eFuse Used During Flash Encryption Process*.
- Generate a random key with espsecure.py:

```
espsecure.py generate_flash_encryption_key my_flash_encryption_key.bin
```

- Burn the key to the device (one time only). **This must be done before first encrypted boot**, otherwise the ESP32 will generate a random key that software can't access or modify:

```
espefuse.py --port PORT burn_key flash_encryption my_flash_encryption_key.bin
```

- Enable flash encryption support in second stage bootloader. In *Project Configuration Menu*, navigate to “Security Features” .
- Select *Enable flash encryption on boot*.
- By default the mode is set for **Development**.
- Select appropriate Bootloader log verbosity under Bootloader config.
- Update to the partition table offset may be required since after enabling flash encryption the size of bootloader is increased. See *Bootloader Size*
- Save the configuration and exit.

Build and flash the complete image including: bootloader, partition table and app. These partitions are initially written to the flash unencrypted

```
idf.py flash monitor
```

On next boot second stage bootloader will encrypt the flash app partition and then reset. Now the sample application would get decrypted at runtime and executed.

At this stage if user wants to update new plaintext application image to flash they should issue following command

```
idf.py encrypted-app-flash monitor
```

For reprogramming all partitions in encrypted format follow *Encrypt Multiple Partitions*.

Release Mode

In Release mode UART bootloader can not perform flash encryption operations and new plaintext images can be downloaded ONLY using OTA scheme which will encrypt the plaintext image before writing to flash.

- Ensure you have a ESP32 device with default flash encryption eFuse settings as shown in *eFuse Used During Flash Encryption Process*.
- Enable flash encryption support in second stage bootloader. In *Project Configuration Menu*, navigate to “Security Features” .
- Select *Enable flash encryption on boot*.
- Select **Release Mode**, by default the mode is set for **Development**. Please note **once the Release mode is selected the ``download_dis_encrypt`` and ``download_dis_decrypt`` eFuse bits will be programmed to disable UART bootloader access to flash contents**.
- Select appropriate Bootloader log verbosity under Bootloader config.
- Update to the partition table offset may be required since after enabling flash encryption the size of bootloader is increased. See *Bootloader Size*
- Save the configuration and exit.

Build and flash the complete image including: bootloader, partition table and app. These partitions are initially written to the flash unencrypted

```
idf.py flash monitor
```

On next boot second stage bootloader will encrypt the flash app partition and then reset. Now the sample application should execute correctly.

Once the flash encryption is enabled in Release mode the bootloader will write protect the FLASH_CRYPT_CNT eFuse.

For subsequent plaintext update in field *OTA scheme* should be used.

. _flash-encrypt-best-practices:

Best Practices

When using Flash Encryption in production:

- Do not reuse the same flash encryption key between multiple devices. This means that an attacker who copies encrypted data from one device cannot transfer it to a second device.
- When using ESP32 V3, if the UART ROM Download Mode is not needed for a production device then it should be disabled to provide an extra level of protection. Do this by calling `esp_efuse_disable_rom_download_mode()` during application startup. Alternatively, configure the project `CONFIG_ESP32_REV_MIN` level to 3 (targeting ESP32 V3 only) and enable `CONFIG_SECURE_DISABLE_ROM_DL_MODE`. The ability to disable ROM Download Mode is not available on earlier ESP32 versions.
- Enable *Secure Boot* as an extra layer of protection, and to prevent an attacker from selectively corrupting any part of the flash before boot.

Possible Failures

Once flash encryption is enabled and if the FLASH_CRYPT_CNT eFuse value has an odd number of bits set then all the partitions (which are marked with encryption flag) are expected to contain encrypted ciphertext. Below are three typical failure cases if the ESP32 is loaded with plaintext data:

1. In case the bootloader partition is re-updated with plaintext bootloader image the ROM loader will fail to load the bootloader and following type of failure will be displayed:

```
rst:0x3 (SW_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
flash read err, 1000
ets_main.c 371
ets Jun  8 2016 00:22:57

rst:0x7 (TG0WDT_SYS_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
flash read err, 1000
ets_main.c 371
ets Jun  8 2016 00:22:57

rst:0x7 (TG0WDT_SYS_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
flash read err, 1000
ets_main.c 371
ets Jun  8 2016 00:22:57

rst:0x7 (TG0WDT_SYS_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
flash read err, 1000
ets_main.c 371
ets Jun  8 2016 00:22:57

rst:0x7 (TG0WDT_SYS_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
flash read err, 1000
```

(continues on next page)

(continued from previous page)

```
ets_main.c 371
ets Jun  8 2016 00:22:57
```

Note: This error also appears in the flash contents is erased or corrupted.

2. In case the bootloader is encrypted but partition table is re-updated with plaintext partition table image the bootloader will fail to read the partition table and following type of failure will be displayed:

```
rst:0x3 (SW_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0018,len:4
load:0x3fff001c,len:10464
ho 0 tail 12 room 4
load:0x40078000,len:19168
load:0x40080400,len:6664
entry 0x40080764
I (60) boot: ESP-IDF v4.0-dev-763-g2c55fae6c-dirty 2nd stage bootloader
I (60) boot: compile time 19:15:54
I (62) boot: Enabling RNG early entropy source...
I (67) boot: SPI Speed      : 40MHz
I (72) boot: SPI Mode      : DIO
I (76) boot: SPI Flash Size : 4MB
E (80) flash_parts: partition 0 invalid magic number 0x94f6
E (86) boot: Failed to verify partition table
E (91) boot: load partition table error!
```

3. In case the bootloader & partition table are encrypted but application is re-updated with plaintext application image the bootloader will fail load the new application and following type of failure will be displayed:

```
rst:0x3 (SW_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0018,len:4
load:0x3fff001c,len:8452
load:0x40078000,len:13616
load:0x40080400,len:6664
entry 0x40080764
I (56) boot: ESP-IDF v4.0-dev-850-gc4447462d-dirty 2nd stage bootloader
I (56) boot: compile time 15:37:14
I (58) boot: Enabling RNG early entropy source...
I (64) boot: SPI Speed      : 40MHz
I (68) boot: SPI Mode      : DIO
I (72) boot: SPI Flash Size : 4MB
I (76) boot: Partition Table:
I (79) boot:  ## Label                Usage                Type ST Offset   Length
I (87) boot:  0 nvs                   WiFi data            01 02 0000a000 00006000
I (94) boot:  1 phy_init              RF data              01 01 00010000 00001000
I (102) boot:  2 factory              factory app          00 00 00020000 00100000
I (109) boot: End of partition table
E (113) esp_image: image at 0x20000 has invalid magic byte
W (120) esp_image: image at 0x20000 has invalid SPI mode 108
W (126) esp_image: image at 0x20000 has invalid SPI size 11
E (132) boot: Factory app partition is not bootable
E (138) boot: No bootable app partitions in the partition table
```

5.15.5 Key Points About Flash Encryption

- The contents of the flash are encrypted using AES-256. The flash encryption key is stored in eFuse internal to the chip, and is (by default) protected from software access.
- The *flash encryption algorithm* is AES-256, where the key is “tweaked” with the offset address of each 32 byte block of flash. This means every 32 byte block (two consecutive 16 byte AES blocks) is encrypted with a unique key derived from the flash encryption key.
- Flash access is transparent via the flash cache mapping feature of ESP32 - any flash regions which are mapped to the address space will be transparently decrypted when read.
 - It may be desirable for some data partitions to remain unencrypted for ease of access, or to use flash-friendly update algorithms that are ineffective if the data is encrypted. NVS partitions for non-volatile storage cannot be encrypted since NVS library is not directly compatible with flash encryption. Refer to *NVS Encryption* for more details.
- If flash encryption may be enabled, the programmer must take certain precautions when writing code that *uses encrypted flash*.
- If secure boot is enabled, reflashing the bootloader of an encrypted device requires a “Reflashable” secure boot digest (see *Flash Encryption and Secure Boot*).

Note: The bootloader app binary `bootloader.bin` may become too large when both secure boot and flash encryption are enabled. See *Bootloader Size*.

Important: Do not interrupt power to the ESP32 while the first boot encryption pass is running. If power is interrupted, the flash contents will be corrupted and require flashing with unencrypted data again. A reflash like this will not count towards the flashing limit.

5.15.6 Using Encrypted Flash

ESP32 app code can check if flash encryption is currently enabled by calling `esp_flash_encryption_enabled()`. Also, device can identify the flash encryption mode by calling `esp_get_flash_encryption_mode()`.

Once flash encryption is enabled, some care needs to be taken when accessing flash contents from code.

Scope of Flash Encryption

Whenever the `FLASH_CRYPT_CNT` eFuse is set to a value with an odd number of bits set, all flash content which is accessed via the MMU’s flash cache is transparently decrypted. This includes:

- Executable application code in flash (IROM).
- All read-only data stored in flash (DROM).
- Any data accessed via `spi_flash_mmap()`.
- The software bootloader image when it is read by the ROM bootloader.

Important: The MMU flash cache unconditionally decrypts all data. Data which is stored unencrypted in the flash will be “transparently decrypted” via the flash cache and appear to software like random garbage.

Reading Encrypted Flash

To read data without using a flash cache MMU mapping, we recommend using the partition read function `esp_partition_read()`. When using this function, data will only be decrypted when it is read from an encrypted partition. Other partitions will be read unencrypted. In this way, software can access encrypted and non-encrypted flash in the same way.

Data which is read via other SPI read APIs are not decrypted:

- Data read via `spi_flash_read()` is not decrypted.
- Data read via ROM function `SPIRead()` is not decrypted (this function is not supported in esp-idf apps).
- Data stored using the Non-Volatile Storage (NVS) API is always stored and read decrypted from the perspective of flash encryption. It is up to the library to provide encryption feature if required. Refer to [NVS Encryption](#) for more details.

Writing Encrypted Flash

Where possible, we recommend using the partition write function `esp_partition_write`. When using this function, data will only be encrypted when writing to encrypted partitions. Data will be written to other partitions unencrypted. In this way, software can access encrypted and non-encrypted flash in the same way.

The `esp_spi_flash_write` function will write data when the `write_encrypted` parameter is set to true. Otherwise, data will be written unencrypted.

The ROM function `esp_rom_spiflash_write_encrypted` will write encrypted data to flash, the ROM function `SPIWrite` will write unencrypted to flash. (these function are not supported in esp-idf apps).

Because data is encrypted in blocks, the minimum write size for encrypted data is 16 bytes (and the alignment is 16 bytes).

5.15.7 Updating Encrypted Flash

OTA Updates

OTA updates to encrypted partitions will automatically write encrypted, as long as the `esp_partition_write` function is used.

Any app image which will be OTA updated onto a device with flash encryption enabled requires [Enable flash encryption on boot](#) option to be enabled in the app configuration as well, when building the app.

Please refer to [OTA](#) for general information about ESP-IDF OTA updates.

5.15.8 Disabling Flash Encryption

If you've accidentally enabled flash encryption for some reason, the next flash of plaintext data will soft-brick the ESP32 (the device will reboot continuously, printing the error `flash read err, 1000`).

If flash encryption is enabled in Development mode, you can disable flash encryption again by writing `FLASH_CRYPT_CNT` eFuse. This can only be done three times per chip.

- First, open [Project Configuration Menu](#) and disable [Enable flash encryption boot](#) under “Security Features” .
- Exit `menuconfig` and save the new configuration.
- Run `idf.py menuconfig` again and double-check you really disabled this option! *If this option is left enabled, the bootloader will immediately re-enable encryption when it boots.*
- Run `idf.py flash` to build and flash a new bootloader and app, without flash encryption enabled.
- **Run `espefuse.py` (in `components/esptool_py/esptool`) to disable the `FLASH_CRYPT_CNT`:**
`espefuse.py burn_efuse FLASH_CRYPT_CNT`

Reset the ESP32 and flash encryption should be disabled, the bootloader will boot as normal.

5.15.9 Limitations of Flash Encryption

Flash encryption protects firmware against unauthorised readout and modification. It is important to understand the limitations of the flash encryption feature:

- Flash encryption is only as strong as the key. For this reason, we recommend keys are generated on the device during first boot (default behaviour). If generating keys off-device, ensure proper procedure is followed and don't share the same key between all production devices.
- Not all data is stored encrypted. If storing data on flash, check if the method you are using (library, API, etc.) supports flash encryption.
- Flash encryption does not prevent an attacker from understanding the high-level layout of the flash. This is because the same AES key is used for every pair of adjacent 16 byte AES blocks. When these adjacent 16 byte blocks contain identical content (such as empty or padding areas), these blocks will encrypt to produce matching pairs of encrypted blocks. This may allow an attacker to make high-level comparisons between encrypted devices (i.e. to tell if two devices are probably running the same firmware version).
- For the same reason, an attacker can always tell when a pair of adjacent 16 byte blocks (32 byte aligned) contain two identical 16 byte sequences. Keep this in mind if storing sensitive data on the flash, design your flash storage so this doesn't happen (using a counter byte or some other non-identical value every 16 bytes is sufficient). *NVS Encryption* deals with this and is suitable for many uses.
- Flash encryption alone may not prevent an attacker from modifying the firmware of the device. To prevent unauthorised firmware from running on the device, use flash encryption in combination with *Secure Boot*.

5.15.10 Flash Encryption and Secure Boot

It is recommended to use flash encryption and secure boot together. However, if Secure Boot is enabled then additional restrictions apply to reflashing the device:

- *OTA Updates* are not restricted (provided the new app is signed correctly with the Secure Boot signing key).
- *Plaintext serial flash updates* are only possible if the *Reflashable* Secure Boot mode is selected and a Secure Boot key was pre-generated and burned to the ESP32 (refer to *Secure Boot* docs.). In this configuration, `idf.py bootloader` will produce a pre-digested bootloader and secure boot digest file for flashing at offset 0x0. When following the plaintext serial reflashing steps it is necessary to re-flash this file before flashing other plaintext data.
- *Reflashing via Pregenerated Flash Encryption Key* is still possible, provided the bootloader is not reflashed. Reflashing the bootloader requires the same *Reflashable* option to be enabled in the Secure Boot config.

5.15.11 Advanced Features

The following information is useful for advanced use of flash encryption:

Encrypted Partition Flag

Some partitions are encrypted by default. Otherwise, it is possible to mark any partition as requiring encryption:

In the *partition table* description CSV files, there is a field for flags.

Usually left blank, if you write “encrypted” in this field then the partition will be marked as encrypted in the partition table, and data written here will be treated as encrypted (same as an app partition):

```
# Name, Type, SubType, Offset, Size, Flags
nvs, data, nvs, 0x9000, 0x6000
phy_init, data, phy, 0xf000, 0x1000
factory, app, factory, 0x10000, 1M
secret_data, 0x40, 0x01, 0x20000, 256K, encrypted
```

- None of the default partition tables include any encrypted data partitions.
- It is not necessary to mark “app” partitions as encrypted, they are always treated as encrypted.
- The “encrypted” flag does nothing if flash encryption is not enabled.
- It is possible to mark the optional `phy` partition with `phy_init` data as encrypted, if you wish to protect this data from physical access readout or modification.
- It is not possible to mark the `nvs` partition as encrypted.

Enabling UART Bootloader Encryption/Decryption

By default, on first boot the flash encryption process will burn eFuses `DISABLE_DL_ENCRYPT`, `DISABLE_DL_DECRYPT` and `DISABLE_DL_CACHE`:

- `DISABLE_DL_ENCRYPT` disables the flash encryption operations when running in UART bootloader boot mode.
- `DISABLE_DL_DECRYPT` disables transparent flash decryption when running in UART bootloader mode, even if `FLASH_CRYPT_CNT` is set to enable it in normal operation.
- `DISABLE_DL_CACHE` disables the entire MMU flash cache when running in UART bootloader mode.

It is possible to burn only some of these eFuses, and write-protect the rest (with unset value 0) before the first boot, in order to preserve them. For example:

```
espefuse.py --port PORT burn_efuse DISABLE_DL_DECRYPT
espefuse.py --port PORT write_protect_efuse DISABLE_DL_ENCRYPT
```

(Note that all 3 of these eFuses are disabled via one write protect bit, so write protecting one will write protect all of them. For this reason, it's necessary to set any bits before write-protecting.)

Important: Write protecting these eFuses to keep them unset is not currently very useful, as `esptool.py` does not support reading encrypted flash.

Important: If `DISABLE_DL_DECRYPT` is left unset (0) this effectively makes flash encryption useless, as an attacker with physical access can use UART bootloader mode (with custom stub code) to read out the flash contents.

Setting FLASH_CRYPT_CONFIG

The `FLASH_CRYPT_CONFIG` eFuse determines the number of bits in the flash encryption key which are “tweaked” with the block offset. See *Flash Encryption Algorithm* for details.

First boot of the bootloader always sets this value to the maximum `0xF`.

It is possible to write these eFuse manually, and write protect it before first boot in order to select different tweak values. This is not recommended.

It is strongly recommended to never write protect `FLASH_CRYPT_CONFIG` when it the value is zero. If this eFuse is set to zero, no bits in the flash encryption key are tweaked and the flash encryption algorithm is equivalent to AES ECB mode.

JTAG Debugging

By default, when Flash Encryption is enabled (in either Development or Release mode) then JTAG debugging is disabled via eFuse. The bootloader does this on first boot, at the same time it enables flash encryption.

See *JTAG with Flash Encryption or Secure Boot* for more information about using JTAG Debugging with Flash Encryption.

5.15.12 Technical Details

The following sections provide some reference information about the operation of flash encryption.

Flash Encryption Algorithm

- AES-256 operates on 16 byte blocks of data. The flash encryption engine encrypts and decrypts data in 32 byte blocks, two AES blocks in series.
- The main flash encryption key is stored in eFuse (BLOCK1) and by default is protected from further writes or software readout.
- AES-256 key size is 256 bits (32 bytes), read from eFuse block 1. The hardware AES engine uses the key in reversed byte order to the order stored in the eFuse block.
 - If CODING_SCHEME eFuse is set to 0 (default “None” Coding Scheme) then the eFuse key block is 256 bits and the key is stored as-is (in reversed byte order).
 - If CODING_SCHEME eFuse is set to 1 (3/4 Encoding) then the eFuse key block is 192 bits (in reversed byte order), so overall entropy is reduced. The hardware flash encryption still operates on a 256-bit key, after being read (and un-reversed), the key is extended by $key = key[0:255] + key[64:127]$.
- AES algorithm is used inverted in flash encryption, so the flash encryption “encrypt” operation is AES decrypt and the “decrypt” operation is AES encrypt. This is for performance reasons and does not alter the effectiveness of the algorithm.
- Each 32 byte block (two adjacent 16 byte AES blocks) is encrypted with a unique key. The key is derived from the main flash encryption key in eFuse, XORed with the offset of this block in the flash (a “key tweak”).
- The specific tweak depends on the setting of FLASH_CRYPT_CONFIG eFuse. This is a 4 bit eFuse, where each bit enables XORing of a particular range of the key bits:
 - Bit 1, bits 0-66 of the key are XORed.
 - Bit 2, bits 67-131 of the key are XORed.
 - Bit 3, bits 132-194 of the key are XORed.
 - Bit 4, bits 195-256 of the key are XORed.

It is recommended that FLASH_CRYPT_CONFIG is always left to set the default value *0xF*, so that all key bits are XORed with the block offset. See [Setting FLASH_CRYPT_CONFIG](#) for details.

- The high 19 bits of the block offset (bit 5 to bit 23) are XORed with the main flash encryption key. This range is chosen for two reasons: the maximum flash size is 16MB (24 bits), and each block is 32 bytes so the least significant 5 bits are always zero.
- There is a particular mapping from each of the 19 block offset bits to the 256 bits of the flash encryption key, to determine which bit is XORed with which. See the variable `_FLASH_ENCRYPTION_TWEAK_PATTERN` in the `espsecure.py` source code for the complete mapping.
- To see the full flash encryption algorithm implemented in Python, refer to the `_flash_encryption_operation()` function in the `espsecure.py` source code.

5.16 ESP-IDF FreeRTOS SMP Changes

5.16.1 Overview

The vanilla FreeRTOS is designed to run on a single core. However the ESP32 is dual core containing a Protocol CPU (known as **CPU 0** or **PRO_CPU**) and an Application CPU (known as **CPU 1** or **APP_CPU**). The two cores are identical in practice and share the same memory. This allows the two cores to run tasks interchangeably between them.

The ESP-IDF FreeRTOS is a modified version of vanilla FreeRTOS which supports symmetric multiprocessing (SMP). ESP-IDF FreeRTOS is based on the Xtensa port of FreeRTOS v8.2.0. This guide outlines the major differences between vanilla FreeRTOS and ESP-IDF FreeRTOS. The API reference for vanilla FreeRTOS can be found via <http://www.freertos.org/a00106.html>

For information regarding features that are exclusive to ESP-IDF FreeRTOS, see [ESP-IDF FreeRTOS Additions](#).

Backported Features: Although ESP-IDF FreeRTOS is based on the Xtensa port of FreeRTOS v8.2.0, a number of FreeRTOS v9.0.0 features have been backported to ESP-IDF.

Tasks and Task Creation: Use `xTaskCreatePinnedToCore()` or `xTaskCreateStaticPinnedToCore()` to create tasks in ESP-IDF FreeRTOS. The last parameter of the two functions is `xCoreID`. This parameter specifies which core the task is pinned to. Acceptable values are 0 for **PRO_CPU**, 1 for **APP_CPU**, or `tskNO_AFFINITY` which allows the task to run on both.

Round Robin Scheduling: The ESP-IDF FreeRTOS scheduler will skip tasks when implementing Round-Robin scheduling between multiple tasks in the Ready state that are of the same priority. To avoid this behavior, ensure that those tasks either enter a blocked state, or are distributed across a wider range of priorities.

Scheduler Suspension: Suspending the scheduler in ESP-IDF FreeRTOS will only affect the scheduler on the the calling core. In other words, calling `vTaskSuspendAll()` on **PRO_CPU** will not prevent **APP_CPU** from scheduling, and vice versa. Use critical sections or semaphores instead for simultaneous access protection.

Tick Interrupt Synchronicity: Tick interrupts of **PRO_CPU** and **APP_CPU** are not synchronized. Do not expect to use `vTaskDelay()` or `vTaskDelayUntil()` as an accurate method of synchronizing task execution between the two cores. Use a counting semaphore instead as their context switches are not tied to tick interrupts due to preemption.

Critical Sections & Disabling Interrupts: In ESP-IDF FreeRTOS, critical sections are implemented using mutexes. Entering critical sections involve taking a mutex, then disabling the scheduler and interrupts of the calling core. However the other core is left unaffected. If the other core attempts to take same mutex, it will spin until the calling core has released the mutex by exiting the critical section.

Floating Point Arithmetic: The ESP32 supports hardware acceleration of single precision floating point arithmetic (`float`). However the use of hardware acceleration leads to some behavioral restrictions in ESP-IDF FreeRTOS. Therefore, tasks that utilize `float` will automatically be pinned to a core if not done so already. Furthermore, `float` cannot be used in interrupt service routines.

Task Deletion: Task deletion behavior has been backported from FreeRTOS v9.0.0 and modified to be SMP compatible. Task memory will be freed immediately when `vTaskDelete()` is called to delete a task that is not currently running and not pinned to the other core. Otherwise, freeing of task memory will still be delegated to the Idle Task.

Thread Local Storage Pointers & Deletion Callbacks: ESP-IDF FreeRTOS has backported the Thread Local Storage Pointers (TLSP) feature. However the extra feature of Deletion Callbacks has been added. Deletion callbacks are called automatically during task deletion and are used to free memory pointed to by TLSP. Call `vTaskSetThreadLocalStoragePointerAndDelCallback()` to set TLSP and Deletion Callbacks.

Configuring ESP-IDF FreeRTOS: Several aspects of ESP-IDF FreeRTOS can be set in the project configuration (`idf.py menuconfig`) such as running ESP-IDF in Unicore (single core) Mode, or configuring the number of Thread Local Storage Pointers each task will have.

5.16.2 Backported Features

The following features have been backported from FreeRTOS v9.0.0 to ESP-IDF.

Static Allocation

This feature has been backported from FreeRTOS v9.0.0 to ESP-IDF. The `CONFIG_FREERTOS_SUPPORT_STATIC_ALLOCATION` option must be enabled in `menuconfig` in order for static allocation functions to be available. Once enabled, the following functions can be called...

- `xTaskCreateStatic()` (see *Backporting Notes* below)
- `xQueueCreateStatic`
- `xSemaphoreCreateBinaryStatic`
- `xSemaphoreCreateCountingStatic`
- `xSemaphoreCreateMutexStatic`
- `xSemaphoreCreateRecursiveMutexStatic`
- `xTimerCreateStatic()` (see *Backporting Notes* below)
- `xEventGroupCreateStatic()`

Other Features

- `vTaskSetThreadLocalStoragePointer()` (see *Backporting Notes* below)
- `pvTaskGetThreadLocalStoragePointer()` (see *Backporting Notes* below)
- `vTimerSetTimerID()`

- `xTimerGetPeriod()`
- `xTimerGetExpiryTime()`
- `pcQueueGetName()`
- `uxSemaphoreGetCount`

Backporting Notes

1) `xTaskCreateStatic()` has been made SMP compatible in a similar fashion to `xTaskCreate()` (see *Tasks and Task Creation*). Therefore `xTaskCreateStaticPinnedToCore()` can also be called.

2) Although vanilla FreeRTOS allows the Timer feature's daemon task to be statically allocated, the daemon task is always dynamically allocated in ESP-IDF. Therefore `vApplicationGetTimerTaskMemory` **does not** need to be defined when using statically allocated timers in ESP-IDF FreeRTOS.

3) The Thread Local Storage Pointer feature has been modified in ESP-IDF FreeRTOS to include Deletion Callbacks (see *Thread Local Storage Pointers & Deletion Callbacks*). Therefore the function `vTaskSetThreadLocalStoragePointerAndDelCallback()` can also be called.

5.16.3 Tasks and Task Creation

Tasks in ESP-IDF FreeRTOS are designed to run on a particular core, therefore two new task creation functions have been added to ESP-IDF FreeRTOS by appending `PinnedToCore` to the names of the task creation functions in vanilla FreeRTOS. The vanilla FreeRTOS functions of `xTaskCreate()` and `xTaskCreateStatic()` have led to the addition of `xTaskCreatePinnedToCore()` and `xTaskCreateStaticPinnedToCore()` in ESP-IDF FreeRTOS (see *Backported Features*).

For more details see [freertos/task.c](#)

The ESP-IDF FreeRTOS task creation functions are nearly identical to their vanilla counterparts with the exception of the extra parameter known as `xCoreID`. This parameter specifies the core on which the task should run on and can be one of the following values.

- 0 pins the task to **PRO_CPU**
- 1 pins the task to **APP_CPU**
- `tskNO_AFFINITY` allows the task to be run on both CPUs

For example `xTaskCreatePinnedToCore(tsk_callback, "APP_CPU Task", 1000, NULL, 10, NULL, 1)` creates a task of priority 10 that is pinned to **APP_CPU** with a stack size of 1000 bytes. It should be noted that the `uxStackDepth` parameter in vanilla FreeRTOS specifies a task's stack depth in terms of the number of words, whereas ESP-IDF FreeRTOS specifies the stack depth in terms of bytes.

Note that the vanilla FreeRTOS functions `xTaskCreate()` and `xTaskCreateStatic()` have been defined in ESP-IDF FreeRTOS as inline functions which call `xTaskCreatePinnedToCore()` and `xTaskCreateStaticPinnedToCore()` respectively with `tskNO_AFFINITY` as the `xCoreID` value.

Each Task Control Block (TCB) in ESP-IDF stores the `xCoreID` as a member. Hence when each core calls the scheduler to select a task to run, the `xCoreID` member will allow the scheduler to determine if a given task is permitted to run on the core that called it.

5.16.4 Scheduling

The vanilla FreeRTOS implements scheduling in the `vTaskSwitchContext()` function. This function is responsible for selecting the highest priority task to run from a list of tasks in the Ready state known as the Ready Tasks List (described in the next section). In ESP-IDF FreeRTOS, each core will call `vTaskSwitchContext()` independently to select a task to run from the Ready Tasks List which is shared between both cores. There are several differences in scheduling behavior between vanilla and ESP-IDF FreeRTOS such as differences in Round Robin scheduling, scheduler suspension, and tick interrupt synchronicity.

Round Robin Scheduling

Given multiple tasks in the Ready state and of the same priority, vanilla FreeRTOS implements Round Robin scheduling between each task. This will result in running those tasks in turn each time the scheduler is called (e.g. every tick interrupt). On the other hand, the ESP-IDF FreeRTOS scheduler may skip tasks when Round Robin scheduling multiple Ready state tasks of the same priority.

The issue of skipping tasks during Round Robin scheduling arises from the way the Ready Tasks List is implemented in FreeRTOS. In vanilla FreeRTOS, `pxReadyTasksList` is used to store a list of tasks that are in the Ready state. The list is implemented as an array of length `configMAX_PRIORITIES` where each element of the array is a linked list. Each linked list is of type `List_t` and contains TCBs of tasks of the same priority that are in the Ready state. The following diagram illustrates the `pxReadyTasksList` structure.

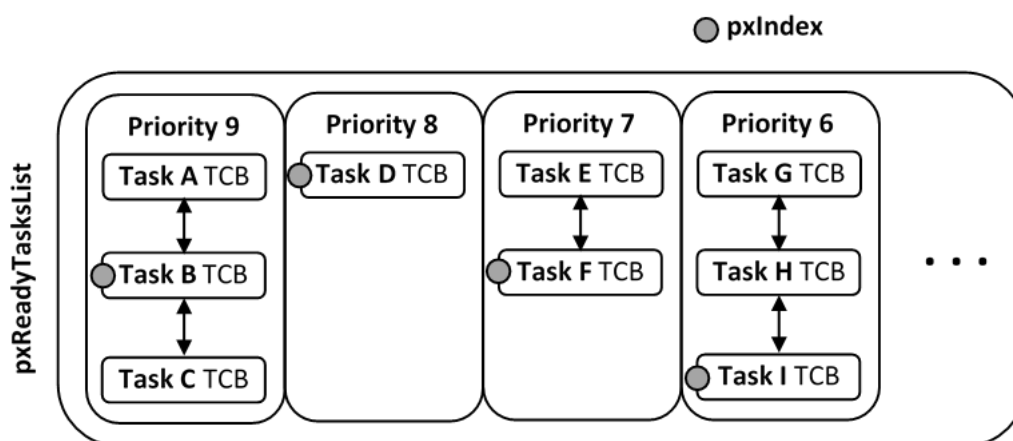


Fig. 33: Illustration of FreeRTOS Ready Task List Data Structure

Each linked list also contains a `pxIndex` which points to the last TCB returned when the list was queried. This index allows the `vTaskSwitchContext()` to start traversing the list at the TCB immediately after `pxIndex` hence implementing Round Robin Scheduling between tasks of the same priority.

In ESP-IDF FreeRTOS, the Ready Tasks List is shared between cores hence `pxReadyTasksList` will contain tasks pinned to different cores. When a core calls the scheduler, it is able to look at the `xCoreID` member of each TCB in the list to determine if a task is allowed to run on calling the core. The ESP-IDF FreeRTOS `pxReadyTasksList` is illustrated below.

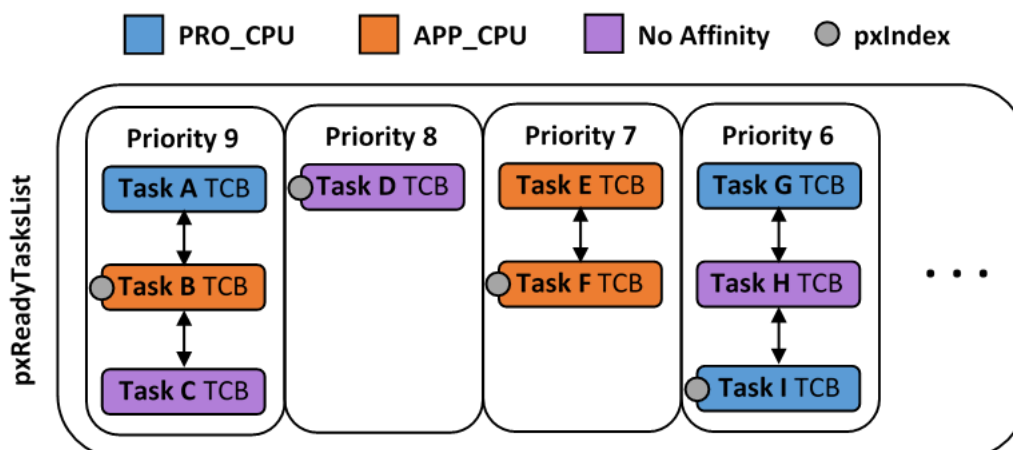


Fig. 34: Illustration of FreeRTOS Ready Task List Data Structure in ESP-IDF

Therefore when **PRO_CPU** calls the scheduler, it will only consider the tasks in blue or purple. Whereas when **APP_CPU** calls the scheduler, it will only consider the tasks in orange or purple.

Although each TCB has an `xCoreID` in ESP-IDF FreeRTOS, the linked list of each priority only has a single `pxIndex`. Therefore when the scheduler is called from a particular core and traverses the linked list, it will skip all TCBs pinned to the other core and point the `pxIndex` at the selected task. If the other core then calls the scheduler, it will traverse the linked list starting at the TCB immediately after `pxIndex`. Therefore, TCBs skipped on the previous scheduler call from the other core would not be considered on the current scheduler call. This issue is demonstrated in the following illustration.

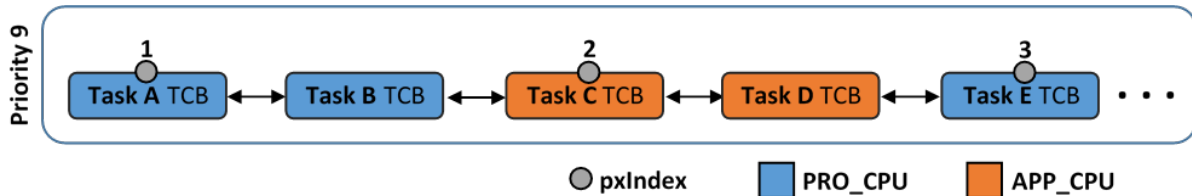


Fig. 35: Illustration of `pxIndex` behavior in ESP-IDF FreeRTOS

Referring to the illustration above, assume that priority 9 is the highest priority, and none of the tasks in priority 9 will block hence will always be either in the running or Ready state.

- 1) **PRO_CPU** calls the scheduler and selects Task A to run, hence moves `pxIndex` to point to Task A
- 2) **APP_CPU** calls the scheduler and starts traversing from the task after `pxIndex` which is Task B. However Task B is not selected to run as it is not pinned to **APP_CPU** hence it is skipped and Task C is selected instead. `pxIndex` now points to Task C
- 3) **PRO_CPU** calls the scheduler and starts traversing from Task D. It skips Task D and selects Task E to run and points `pxIndex` to Task E. Notice that Task B isn't traversed because it was skipped the last time **APP_CPU** called the scheduler to traverse the list.
- 4) The same situation with Task D will occur if **APP_CPU** calls the scheduler again as `pxIndex` now points to Task E

One solution to the issue of task skipping is to ensure that every task will enter a blocked state so that they are removed from the Ready Task List. Another solution is to distribute tasks across multiple priorities such that a given priority will not be assigned multiple tasks that are pinned to different cores.

Scheduler Suspension

In vanilla FreeRTOS, suspending the scheduler via `vTaskSuspendAll()` will prevent calls of `vTaskSwitchContext` from context switching until the scheduler has been resumed with `xTaskResumeAll()`. However servicing ISRs are still permitted. Therefore any changes in task states as a result from the current running task or ISRs will not be executed until the scheduler is resumed. Scheduler suspension in vanilla FreeRTOS is a common protection method against simultaneous access of data shared between tasks, whilst still allowing ISRs to be serviced.

In ESP-IDF FreeRTOS, `xTaskSuspendAll()` will only prevent calls of `vTaskSwitchContext()` from switching contexts on the core that called for the suspension. Hence if **PRO_CPU** calls `vTaskSuspendAll()`, **APP_CPU** will still be able to switch contexts. If data is shared between tasks that are pinned to different cores, scheduler suspension is **NOT** a valid method of protection against simultaneous access. Consider using critical sections (disables interrupts) or semaphores (does not disable interrupts) instead when protecting shared resources in ESP-IDF FreeRTOS.

In general, it's better to use other RTOS primitives like mutex semaphores to protect against data shared between tasks, rather than `vTaskSuspendAll()`.

Tick Interrupt Synchronicity

In ESP-IDF FreeRTOS, tasks on different cores that unblock on the same tick count might not run at exactly the same time due to the scheduler calls from each core being independent, and the tick interrupts to each core being unsynchronized.

In vanilla FreeRTOS the tick interrupt triggers a call to `xTaskIncrementTick()` which is responsible for incrementing the tick counter, checking if tasks which have called `vTaskDelay()` have fulfilled their delay period, and moving those tasks from the Delayed Task List to the Ready Task List. The tick interrupt will then call the scheduler if a context switch is necessary.

In ESP-IDF FreeRTOS, delayed tasks are unblocked with reference to the tick interrupt on PRO_CPU as PRO_CPU is responsible for incrementing the shared tick count. However tick interrupts to each core might not be synchronized (same frequency but out of phase) hence when PRO_CPU receives a tick interrupt, APP_CPU might not have received it yet. Therefore if multiple tasks of the same priority are unblocked on the same tick count, the task pinned to PRO_CPU will run immediately whereas the task pinned to APP_CPU must wait until APP_CPU receives its out of sync tick interrupt. Upon receiving the tick interrupt, APP_CPU will then call for a context switch and finally switches contexts to the newly unblocked task.

Therefore, task delays should **NOT** be used as a method of synchronization between tasks in ESP-IDF FreeRTOS. Instead, consider using a counting semaphore to unblock multiple tasks at the same time.

5.16.5 Critical Sections & Disabling Interrupts

Vanilla FreeRTOS implements critical sections in `vTaskEnterCritical` which disables the scheduler and calls `portDISABLE_INTERRUPTS`. This prevents context switches and servicing of ISRs during a critical section. Therefore, critical sections are used as a valid protection method against simultaneous access in vanilla FreeRTOS.

On the other hand, the ESP32 has no hardware method for cores to disable each other's interrupts. Calling `portDISABLE_INTERRUPTS()` will have no effect on the interrupts of the other core. Therefore, disabling interrupts is **NOT** a valid protection method against simultaneous access to shared data as it leaves the other core free to access the data even if the current core has disabled its own interrupts.

For this reason, ESP-IDF FreeRTOS implements critical sections using mutexes, and calls to enter or exit a critical must provide a mutex that is associated with a shared resource requiring access protection. When entering a critical section in ESP-IDF FreeRTOS, the calling core will disable its scheduler and interrupts similar to the vanilla FreeRTOS implementation. However, the calling core will also take the mutex whilst the other core is left unaffected during the critical section. If the other core attempts to take the same mutex, it will spin until the mutex is released. Therefore, the ESP-IDF FreeRTOS implementation of critical sections allows a core to have protected access to a shared resource without disabling the other core. The other core will only be affected if it tries to concurrently access the same resource.

The ESP-IDF FreeRTOS critical section functions have been modified as follows...

- `taskENTER_CRITICAL(mux)`, `taskENTER_CRITICAL_ISR(mux)`, `portENTER_CRITICAL(mux)`, `portENTER_CRITICAL_ISR(mux)` are all macro defined to call `vTaskEnterCritical()`
- `taskEXIT_CRITICAL(mux)`, `taskEXIT_CRITICAL_ISR(mux)`, `portEXIT_CRITICAL(mux)`, `portEXIT_CRITICAL_ISR(mux)` are all macro defined to call `vTaskExitCritical()`
- `portENTER_CRITICAL_SAFE(mux)`, `portEXIT_CRITICAL_SAFE(mux)` macro identifies the context of execution, i.e. ISR or Non-ISR, and calls appropriate critical section functions (`port*_CRITICAL` in Non-ISR and `port*_CRITICAL_ISR` in ISR) in order to be in compliance with Vanilla FreeRTOS.

For more details see [freertos/include/freertos/portmacro.h](#) and [freertos/task.c](#)

It should be noted that when modifying vanilla FreeRTOS code to be ESP-IDF FreeRTOS compatible, it is trivial to modify the type of critical section called as they are all defined to call the same function. As long as the same mutex is provided upon entering and exiting, the type of call should not matter.

5.16.6 Floating Point Arithmetic

The ESP32 supports hardware acceleration of single precision floating point arithmetic (`float`) via Floating Point Units (FPU, also known as coprocessors) attached to each core. The use of the FPUs imposes some behavioral restrictions on ESP-IDF FreeRTOS.

ESP-IDF FreeRTOS implements Lazy Context Switching for FPUs. In other words, the state of a core's FPU registers are not immediately saved when a context switch occurs. Therefore, tasks that utilize `float` must be pinned to a particular core upon creation. If not, ESP-IDF FreeRTOS will automatically pin the task in question to whichever core the task was running on upon the task's first use of `float`. Likewise due to Lazy Context Switching, interrupt service routines must also not use `float`.

ESP32 does not support hardware acceleration for double precision floating point arithmetic (`double`). Instead `double` is implemented via software hence the behavioral restrictions with regards to `float` do not apply to `double`. Note that due to the lack of hardware acceleration, `double` operations may consume significantly larger amount of CPU time in comparison to `float`.

5.16.7 Task Deletion

FreeRTOS task deletion prior to v9.0.0 delegated the freeing of task memory entirely to the Idle Task. Currently, the freeing of task memory will occur immediately (within `vTaskDelete()`) if the task being deleted is not currently running or is not pinned to the other core (with respect to the core `vTaskDelete()` is called on). TLSP deletion callbacks will also run immediately if the same conditions are met.

However, calling `vTaskDelete()` to delete a task that is either currently running or pinned to the other core will still result in the freeing of memory being delegated to the Idle Task.

5.16.8 Thread Local Storage Pointers & Deletion Callbacks

Thread Local Storage Pointers (TLSP) are pointers stored directly in the TCB. TLSP allow each task to have its own unique set of pointers to data structures. However task deletion behavior in vanilla FreeRTOS does not automatically free the memory pointed to by TLSP. Therefore if the memory pointed to by TLSP is not explicitly freed by the user before task deletion, memory leak will occur.

ESP-IDF FreeRTOS provides the added feature of Deletion Callbacks. Deletion Callbacks are called automatically during task deletion to free memory pointed to by TLSP. Each TLSP can have its own Deletion Callback. Note that due to the [Task Deletion](#) behavior, there can be instances where Deletion Callbacks are called in the context of the Idle Tasks. Therefore Deletion Callbacks **should never attempt to block** and critical sections should be kept as short as possible to minimize priority inversion.

Deletion callbacks are of type `void (*TlsDeleteCallbackFunction_t)(int, void *)` where the first parameter is the index number of the associated TLSP, and the second parameter is the TLSP itself.

Deletion callbacks are set alongside TLSP by calling `vTaskSetThreadLocalStoragePointerAndDeleteCallback()`. Calling the vanilla FreeRTOS function `vTaskSetThreadLocalStoragePointer()` will simply set the TLSP's associated Deletion Callback to `NULL` meaning that no callback will be called for that TLSP during task deletion. If a deletion callback is `NULL`, users should manually free the memory pointed to by the associated TLSP before task deletion in order to avoid memory leak.

`CONFIG_FREERTOS_THREAD_LOCAL_STORAGE_POINTERS` in `menuconfig` can be used to configure the number TLSP and Deletion Callbacks a TCB will have.

For more details see [FreeRTOS API reference](#).

5.16.9 Configuring ESP-IDF FreeRTOS

The ESP-IDF FreeRTOS can be configured in the project configuration menu (`idf.py menuconfig`) under `Component Config/FreeRTOS`. The following section highlights some of the ESP-IDF FreeRTOS configuration options. For a full list of ESP-IDF FreeRTOS configurations, see [FreeRTOS](#)

`CONFIG_FREERTOS_UNICORE` will run ESP-IDF FreeRTOS only on `PRO_CPU`. Note that this is **not equivalent to running vanilla FreeRTOS**. Behaviors of multiple components in ESP-IDF will be modified such as `esp32/cpu_start.c`. For more details regarding the effects of running ESP-IDF FreeRTOS on a single core, search for occurrences of `CONFIG_FREERTOS_UNICORE` in the ESP-IDF components.

`CONFIG_FREERTOS_THREAD_LOCAL_STORAGE_POINTERS` will define the number of Thread Local Storage Pointers each task will have in ESP-IDF FreeRTOS.

`CONFIG_FREERTOS_SUPPORT_STATIC_ALLOCATION` will enable the backported functionality of `xTaskCreateStaticPinnedToCore()` in ESP-IDF FreeRTOS

`CONFIG_FREERTOS_ASSERT_ON_UNTESTED_FUNCTION` will trigger a halt in particular functions in ESP-IDF FreeRTOS which have not been fully tested in an SMP context.

`CONFIG_FREERTOS_TASK_FUNCTION_WRAPPER` will enclose all task functions within a wrapper function. In the case that a task function mistakenly returns (i.e. does not call `vTaskDelete()`), the call flow will return to the wrapper function. The wrapper function will then log an error and abort the application, as illustrated below:

```
E (25) FreeRTOS: FreeRTOS task should not return. Aborting now!
abort() was called at PC 0x40085c53 on core 0
```

5.17 General Notes About ESP-IDF Programming

5.17.1 Application startup flow

This note explains various steps which happen before `app_main` function of an ESP-IDF application is called.

The high level view of startup process is as follows:

1. First-stage bootloader in ROM loads second-stage bootloader image to RAM (IRAM & DRAM) from flash offset 0x1000.
2. Second-stage bootloader loads partition table and main app image from flash. Main app incorporates both RAM segments and read-only segments mapped via flash cache.
3. Main app image executes. At this point the second CPU and RTOS scheduler can be started.

This process is explained in detail in the following sections.

First stage bootloader

After SoC reset, PRO CPU will start running immediately, executing reset vector code, while APP CPU will be held in reset. During startup process, PRO CPU does all the initialization. APP CPU reset is de-asserted in the `call_start_cpu0` function of application startup code. Reset vector code is located at address 0x40000400 in the mask ROM of the ESP32 chip and can not be modified.

Startup code called from the reset vector determines the boot mode by checking `GPIO_STRAP_REG` register for bootstrap pin states. Depending on the reset reason, the following takes place:

1. Reset from deep sleep: if the value in `RTC_CNTL_STORE6_REG` is non-zero, and CRC value of RTC memory in `RTC_CNTL_STORE7_REG` is valid, use `RTC_CNTL_STORE6_REG` as an entry point address and jump immediately to it. If `RTC_CNTL_STORE6_REG` is zero, or `RTC_CNTL_STORE7_REG` contains invalid CRC, or once the code called via `RTC_CNTL_STORE6_REG` returns, proceed with boot as if it was a power-on reset. **Note:** to run customized code at this point, a deep sleep stub mechanism is provided. Please see [deep sleep](#) documentation for this.
2. For power-on reset, software SOC reset, and watchdog SOC reset: check the `GPIO_STRAP_REG` register if UART or SDIO download mode is requested. If this is the case, configure UART or SDIO, and wait for code to be downloaded. Otherwise, proceed with boot as if it was due to software CPU reset.
3. For software CPU reset and watchdog CPU reset: configure SPI flash based on EFUSE values, and attempt to load the code from flash. This step is described in more detail in the next paragraphs. If loading code from flash fails, unpack BASIC interpreter into the RAM and start it. Note that RTC watchdog is still enabled when this happens, so unless any input is received by the interpreter, watchdog will reset the SOC in a few hundred milliseconds, repeating the whole process. If the interpreter receives any input from the UART, it disables the watchdog.

Application binary image is loaded from flash starting at address 0x1000. First 4kB sector of flash is used to store secure boot IV and signature of the application image. Please check secure boot documentation for details about this.

Second stage bootloader

In ESP-IDF, the binary image which resides at offset 0x1000 in flash is the second stage bootloader. Second stage bootloader source code is available in `components/bootloader` directory of ESP-IDF. Note that this arrangement is not the only one possible with the ESP32 chip. It is possible to write a fully featured application which would work when flashed to offset 0x1000, but this is out of scope of this document. Second stage bootloader is used in ESP-IDF to add flexibility to flash layout (using partition tables), and allow for various flows associated with flash encryption, secure boot, and over-the-air updates (OTA) to take place.

When the first stage bootloader is finished checking and loading the second stage bootloader, it jumps to the second stage bootloader entry point found in the binary image header.

Second stage bootloader reads the partition table found at offset 0x8000. See [partition tables](#) documentation for more information. The bootloader finds factory and OTA partitions, and decides which one to boot based on data found in `OTA info` partition.

For the selected partition, second stage bootloader copies data and code sections which are mapped into IRAM and DRAM to their load addresses. For sections which have load addresses in DROM and IROM regions, flash MMU is configured to provide the correct mapping. Note that the second stage bootloader configures flash MMU for both PRO and APP CPUs, but it only enables flash MMU for PRO CPU. Reason for this is that second stage bootloader code is loaded into the memory region used by APP CPU cache. The duty of enabling cache for APP CPU is passed on to the application. Once code is loaded and flash MMU is set up, second stage bootloader jumps to the application entry point found in the binary image header.

Currently it is not possible to add application-defined hooks to the bootloader to customize application partition selection logic. This may be required to load different application image depending on a state of a GPIO, for example. Such customization features will be added to ESP-IDF in the future. For now, bootloader can be customized by copying bootloader component into application directory and making necessary changes there. ESP-IDF build system will compile the component in application directory instead of ESP-IDF components directory in this case.

Application startup

ESP-IDF application entry point is `call_start_cpu0` function found in `components/esp32/cpu_start.c`. Two main things this function does are to enable heap allocator and to make APP CPU jump to its entry point, `call_start_cpu1`. The code on PRO CPU sets the entry point for APP CPU, de-asserts APP CPU reset, and waits for a global flag to be set by the code running on APP CPU, indicating that it has started. Once this is done, PRO CPU jumps to `start_cpu0` function, and APP CPU jumps to `start_cpu1` function.

Both `start_cpu0` and `start_cpu1` are weak functions, meaning that they can be overridden in the application, if some application-specific change to initialization sequence is needed. Default implementation of `start_cpu0` enables or initializes components depending on choices made in `menuconfig`. Please see source code of this function in `components/esp32/cpu_start.c` for an up to date list of steps performed. Note that any C++ global constructors present in the application will be called at this stage. Once all essential components are initialized, `main task` is created and FreeRTOS scheduler is started.

While PRO CPU does initialization in `start_cpu0` function, APP CPU spins in `start_cpu1` function, waiting for the scheduler to be started on the PRO CPU. Once the scheduler is started on the PRO CPU, code on the APP CPU starts the scheduler as well.

Main task is the task which runs `app_main` function. Main task stack size and priority can be configured in `menuconfig`. Application can use this task for initial application-specific setup, for example to launch other tasks. Application can also use main task for event loops and other general purpose activities. If `app_main` function returns, main task is deleted.

5.17.2 Application memory layout

ESP32 chip has flexible memory mapping features. This section describes how ESP-IDF uses these features by default.

Application code in ESP-IDF can be placed into one of the following memory regions.

IRAM (instruction RAM)

ESP-IDF allocates part of *Internal SRAM0* region (defined in the Technical Reference Manual) for instruction RAM. Except for the first 64 kB block which is used for PRO and APP CPU caches, the rest of this memory range (i.e. from 0x40080000 to 0x400A0000) is used to store parts of application which need to run from RAM.

A few components of ESP-IDF and parts of WiFi stack are placed into this region using the linker script.

If some application code needs to be placed into IRAM, it can be done using `IRAM_ATTR` define:

```
#include "esp_attr.h"

void IRAM_ATTR gpio_isr_handler(void* arg)
{
    // ...
}
```

Here are the cases when parts of application may or should be placed into IRAM.

- Interrupt handlers must be placed into IRAM if `ESP_INTR_FLAG_IRAM` is used when registering the interrupt handler. In this case, ISR may only call functions placed into IRAM or functions present in ROM. *Note 1:* all FreeRTOS APIs are currently placed into IRAM, so are safe to call from interrupt handlers. If the ISR is placed into IRAM, all constant data used by the ISR and functions called from ISR (including, but not limited to, `const char` arrays), must be placed into DRAM using `DRAM_ATTR`.
- Some timing critical code may be placed into IRAM to reduce the penalty associated with loading the code from flash. ESP32 reads code and data from flash via a 32 kB cache. In some cases, placing a function into IRAM may reduce delays caused by a cache miss.

IROM (code executed from Flash)

If a function is not explicitly placed into IRAM or RTC memory, it is placed into flash. The mechanism by which Flash MMU is used to allow code execution from flash is described in the Technical Reference Manual. ESP-IDF places the code which should be executed from flash starting from the beginning of 0x400D0000 — 0x40400000 region. Upon startup, second stage bootloader initializes Flash MMU to map the location in flash where code is located into the beginning of this region. Access to this region is transparently cached using two 32kB blocks in 0x40070000 — 0x40080000 range.

Note that the code outside 0x40000000 — 0x40400000 region may not be reachable with Window ABI `CALLX` instructions, so special care is required if 0x40400000 — 0x40800000 or 0x40800000 — 0x40C00000 regions are used by the application. ESP-IDF doesn't use these regions by default.

RTC fast memory

The code which has to run after wake-up from deep sleep mode has to be placed into RTC memory. Please check detailed description in [deep sleep](#) documentation.

DRAM (data RAM)

Non-constant static data and zero-initialized data is placed by the linker into the 256 kB 0x3FFB0000 — 0x3FFF0000 region. Note that this region is reduced by 64kB (by shifting start address to 0x3FFC0000) if Bluetooth stack is used. Length of this region is also reduced by 16 kB or 32kB if trace memory is used. All space which is left in this region after placing static data there is used for the runtime heap.

Constant data may also be placed into DRAM, for example if it is used in an ISR (see notes in IRAM section above). To do that, `DRAM_ATTR` define can be used:

```
DRAM_ATTR const char[] format_string = "%p %x";
char buffer[64];
sprintf(buffer, format_string, ptr, val);
```

Needless to say, it is not advised to use `printf` and other output functions in ISRs. For debugging purposes, use `ESP_EARLY_LOGx` macros when logging from ISRs. Make sure that both TAG and format string are placed into DRAM in that case.

The macro `__NOINIT_ATTR` can be used as attribute to place data into `.noinit` section. The values placed into this section will not be initialized at startup and keep its value after software restart.

Example:

```
__NOINIT_ATTR uint32_t noinit_data;
```

DROM (data stored in Flash)

By default, constant data is placed by the linker into a 4 MB region (`0x3F400000 — 0x3F800000`) which is used to access external flash memory via Flash MMU and cache. Exceptions to this are literal constants which are embedded by the compiler into application code.

RTC slow memory

Global and static variables used by code which runs from RTC memory (i.e. deep sleep stub code) must be placed into RTC slow memory. Please check detailed description in [deep sleep](#) documentation.

The attribute macro named `RTC_NOINIT_ATTR` can be used to place data into this type of memory. The values placed into this section keep their value after waking from deep sleep.

Example:

```
RTC_NOINIT_ATTR uint32_t rtc_noinit_data;
```

5.17.3 DMA Capable Requirement

Most DMA controllers (e.g. SPI, sdmmc, etc.) have requirements that sending/receiving buffers should be placed in DRAM and word-aligned. We suggest to place DMA buffers in static variables rather than in the stack. Use macro `DMA_ATTR` to declare global/local static variables like:

```
DMA_ATTR uint8_t buffer[]="I want to send something";

void app_main()
{
    // initialization code...
    spi_transaction_t temp = {
        .tx_buffer = buffer,
        .length = 8*sizeof(buffer),
    };
    spi_device_transmit( spi, &temp );
    // other stuff
}
```

Or:

```
void app_main()
{
    DMA_ATTR static uint8_t buffer[]="I want to send something";
    // initialization code...
    spi_transaction_t temp = {
        .tx_buffer = buffer,
        .length = 8*sizeof(buffer),
    };
};
```

(continues on next page)

(continued from previous page)

```

spi_device_transmit( spi, &temp );
// other stuff
}

```

Placing DMA buffers in the stack is still allowed, though you have to keep in mind:

1. Never try to do this if the stack is in the pSRAM. If the stack of a task is placed in the pSRAM, several steps have to be taken as described in [Support for external RAM](#) (at least SPI-RAM_ALLOW_STACK_EXTERNAL_MEMORY option enabled in the menuconfig). Make sure your task is not in the pSRAM.
2. Use macro WORD_ALIGNED_ATTR in functions before variables to place them in proper positions like:

```

void app_main()
{
    uint8_t stuff;
    WORD_ALIGNED_ATTR uint8_t buffer[]="I want to send something"; //or the_
    ↪buffer will be placed right after stuff.
    // initialization code...
    spi_transaction_t temp = {
        .tx_buffer = buffer,
        .length = 8*sizeof(buffer),
    };
    spi_device_transmit( spi, &temp );
    // other stuff
}

```

5.18 High-Level Interrupts

The Xtensa architecture has support for 32 interrupts, divided over 8 levels, plus an assortment of exceptions. On the ESP32, the interrupt mux allows most interrupt sources to be routed to these interrupts using the [interrupt allocator](#). Normally, interrupts will be written in C, but ESP-IDF allows high-level interrupts to be written in assembly as well, allowing for very low interrupt latencies.

5.18.1 Interrupt Levels

| Level | Symbol | Remark |
|-------|-------------------|--|
| 1 | N/A | Exception and level 0 interrupts. Handled by ESP-IDF |
| 2-3 | N/A | Medium level interrupts. Handled by ESP-IDF |
| 4 | xt_highint4 | Normally used by ESP-IDF debug logic |
| 5 | xt_highint5 | Free to use |
| NMI | xt_nmi | Free to use |
| dbg | xt_debugexception | Debug exception. Called on e.g. a BREAK instruction. |

Using these symbols is done by creating an assembly file (suffix .S) and defining the named symbols, like this:

```

.section .iram1,"ax"
.global xt_highint5
.type xt_highint5,@function
.align 4
xt_highint5:
... your code here
rsr a0, EXCSAVE_5
rfi 5

```

For a real-life example, see the [esp32/dport_panic_highint_hdl.S](#) file; the panic handler interrupt is implemented there.

5.18.2 Notes

- Do not call C code from a high-level interrupt; because these interrupts still run in critical sections, this can cause crashes. (The panic handler interrupt does call normal C code, but this is OK because there is no intention of returning to the normal code flow afterwards.)
- Make sure your assembly code gets linked in. If the interrupt handler symbol is the only symbol the rest of the code uses from this file, the linker will take the default ISR instead and not link the assembly file into the final project. To get around this, in the assembly file, define a symbol, like this:

```
.global ld_include_my_isr_file
ld_include_my_isr_file:
```

(The symbol is called `ld_include_my_isr_file` here but can have any arbitrary name not defined anywhere else.) Then, in the component.mk, add this file as an unresolved symbol to the ld command line arguments:

```
COMPONENT_ADD_LDFLAGS := -u ld_include_my_isr_file
```

This should cause the linker to always include a file defining `ld_include_my_isr_file`, causing the ISR to always be linked in.

- High-level interrupts can be routed and handled using `esp_intr_alloc` and associated functions. The handler and handler arguments to `esp_intr_alloc` must be NULL, however.
- In theory, medium priority interrupts could also be handled in this way. For now, ESP-IDF does not support this.

5.19 JTAG Debugging

This document provides a guide to installing OpenOCD for ESP32 and debugging using GDB. The document is structured as follows:

Introduction Introduction to the purpose of this guide.

How it Works? Description how ESP32, JTAG interface, OpenOCD and GDB are interconnected and working together to enable debugging of ESP32.

Selecting JTAG Adapter What are the criteria and options to select JTAG adapter hardware.

Setup of OpenOCD Procedure to install OpenOCD and verify that it is installed.

Configuring ESP32 Target Configuration of OpenOCD software and set up JTAG adapter hardware that will make together a debugging target.

Launching Debugger Steps to start up a debug session with GDB from *Eclipse* and from *Command Line*.

Debugging Examples If you are not familiar with GDB, check this section for debugging examples provided from *Eclipse* as well as from *Command Line*.

Building OpenOCD from Sources Procedure to build OpenOCD from sources for *Windows*, *Linux* and *MacOS* operating systems.

Tips and Quirks This section provides collection of tips and quirks related JTAG debugging of ESP32 with OpenOCD and GDB.

5.19.1 Introduction

The ESP32 has two powerful Xtensa cores, allowing for a great deal of variety of program architectures. The FreeRTOS OS that comes with ESP-IDF is capable of multi-core preemptive multithreading, allowing for an intuitive way of writing software.

The downside of the ease of programming is that debugging without the right tools is harder: figuring out a bug that is caused by two threads, running even simultaneously on two different CPU cores, can take a long time when all you have are printf statements. A better and in many cases quicker way to debug such problems is by using a debugger, connected to the processors over a debug port.

Espressif has ported OpenOCD to support the ESP32 processor and the multicore FreeRTOS, which will be the foundation of most ESP32 apps, and has written some tools to help with features OpenOCD does not support natively.

This document provides a guide to installing OpenOCD for ESP32 and debugging using GDB under Linux, Windows and MacOS. Except for OS specific installation procedures, the s/w user interface and use procedures are the same across all supported operating systems.

Note: Screenshots presented in this document have been made for Eclipse Neon 3 running on Ubuntu 16.04 LTE. There may be some small differences in what a particular user interface looks like, depending on whether you are using Windows, MacOS or Linux and / or a different release of Eclipse.

5.19.2 How it Works?

The key software and hardware to perform debugging of ESP32 with OpenOCD over JTAG (Joint Test Action Group) interface is presented below and includes **xtensa-esp32-elf-gdb debugger**, **OpenOCD on chip debugger** and **JTAG adapter** connected to **ESP32** target.

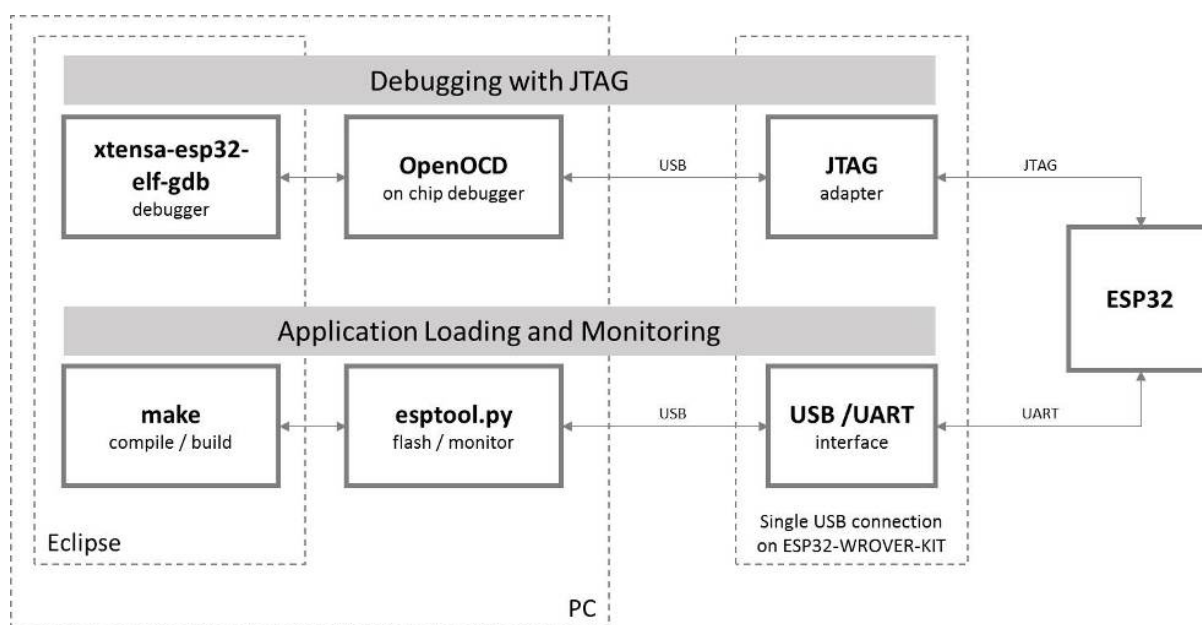


Fig. 36: JTAG debugging - overview diagram

Under “Application Loading and Monitoring” there is another software and hardware to compile, build and flash application to ESP32, as well as to provide means to monitor diagnostic messages from ESP32.

Debugging using JTAG and application loading / monitoring is integrated under the [Eclipse](#) environment, to provide quick and easy transition from writing, compiling and loading the code to debugging, back to writing the code, and so on. All the software is available for Windows, Linux and MacOS platforms.

If the [ESP-WROVER-KIT](#) is used, then connection from PC to ESP32 is done effectively with a single USB cable thanks to FT2232H chip installed on WROVER, which provides two USB channels, one for JTAG and the second for UART connection.

Depending on user preferences, both *debugger* and *idf.py build* can be operated directly from terminal/command line, instead from Eclipse.

5.19.3 Selecting JTAG Adapter

The quickest and most convenient way to start with JTAG debugging is by using [ESP-WROVER-KIT](#). Each version of this development board has JTAG interface already build in. No need for an external JTAG adapter and extra wiring / cable to connect JTAG to ESP32. WROVER KIT is using FT2232H JTAG interface operating at 20 MHz clock speed, which is difficult to achieve with an external adapter.

If you decide to use separate JTAG adapter, look for one that is compatible with both the voltage levels on the ESP32 as well as with the OpenOCD software. The JTAG port on the ESP32 is an industry-standard JTAG port which lacks (and does not need) the TRST pin. The JTAG I/O pins all are powered from the VDD_3P3_RTC pin (which normally would be powered by a 3.3 V rail) so the JTAG adapter needs to be able to work with JTAG pins in that voltage range.

On the software side, OpenOCD supports a fair amount of JTAG adapters. See <http://openocd.org/doc/html/Debug-Adapter-Hardware.html> for an (unfortunately slightly incomplete) list of the adapters OpenOCD works with. This page lists SWD-compatible adapters as well; take note that the ESP32 does not support SWD. JTAG adapters that are hardcoded to a specific product line, e.g. ST-LINK debugging adapters for STM32 families, will not work.

The minimal signalling to get a working JTAG connection are TDI, TDO, TCK, TMS and GND. Some JTAG debuggers also need a connection from the ESP32 power line to a line called e.g. Vtar to set the working voltage. SRST can optionally be connected to the CH_PD of the ESP32, although for now, support in OpenOCD for that line is pretty minimal.

5.19.4 Setup of OpenOCD

If you have already set up ESP-IDF with CMake build system according to the *Getting Started Guide*, then OpenOCD is already installed. After *setting up the environment* in your terminal, you should be able to run OpenOCD. Check this by executing the following command:

```
openocd --version
```

The output should be as follows (although the version may be more recent than listed here):

```
Open On-Chip Debugger v0.10.0-esp32-20190708 (2019-07-08-11:04)
Licensed under GNU GPL v2
For bug reports, read
  http://openocd.org/doc/doxygen/bugs.html
```

You may also verify that OpenOCD knows where its configuration scripts are located by printing the value of OPENOCD_SCRIPTS environment variable, by typing `echo $OPENOCD_SCRIPTS` (for Linux and macOS) or `echo %OPENOCD_SCRIPTS%` (for Windows). If a valid path is printed, then OpenOCD is set up correctly.

If any of these steps do not work, please go back to the *setting up the tools* section of the Getting Started Guide.

Note: It is also possible to build OpenOCD from source. Please refer to *Building OpenOCD from Sources* section for details.

5.19.5 Configuring ESP32 Target

Once OpenOCD is installed, move to configuring ESP32 target (i.e ESP32 board with JTAG interface). You will do it in the following three steps:

- Configure and connect JTAG interface
- Run OpenOCD
- Upload application for debugging

Configure and connect JTAG interface

This step depends on JTAG and ESP32 board you are using - see the two cases described below.

Configure WROVER JTAG Interface All versions of ESP-WROVER-KIT boards have built-in JTAG functionality. Putting it to work requires setting jumpers to enable JTAG functionality, setting SPI flash voltage and configuring USB drivers. Please refer to step by step instructions below.

Configure Hardware

1. Enable on-board JTAG functionality by setting JP8 according to *ESP-WROVER-KIT V4.1 Getting Started Guide*, Section *Setup Options*.
2. Verify if ESP32 pins used for JTAG communication are not connected to some other h/w that may disturb JTAG operation:

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

Configure USB Drivers Install and configure USB drivers, so OpenOCD is able to communicate with JTAG interface on ESP-WROVER-KIT board as well as with UART interface used to upload application for flash. Follow steps below specific to your operating system.

Note: ESP-WROVER-KIT uses an FT2232 adapter. The following instructions can also be used for other FT2232 based JTAG adapters.

Windows

1. Using standard USB A / micro USB B cable connect ESP-WROVER-KIT to the computer. Switch the WROVER KIT on.
2. Wait until USB ports of WROVER KIT are recognized by Windows and drives are installed. If they do not install automatically, then download them from <http://www.ftdichip.com/Drivers/D2XX.htm> and install manually.
3. Download Zadig tool (Zadig_X.X.exe) from <http://zadig.akeo.ie/> and run it.
4. In Zadig tool go to “Options” and check “List All Devices” .
5. Check the list of devices that should contain two WROVER specific USB entries: “Dual RS232-HS (Interface 0)” and “Dual RS232-HS (Interface 1)” . The driver name would be “FTDIBUS (vxxxx)” and USB ID: 0403 6010.

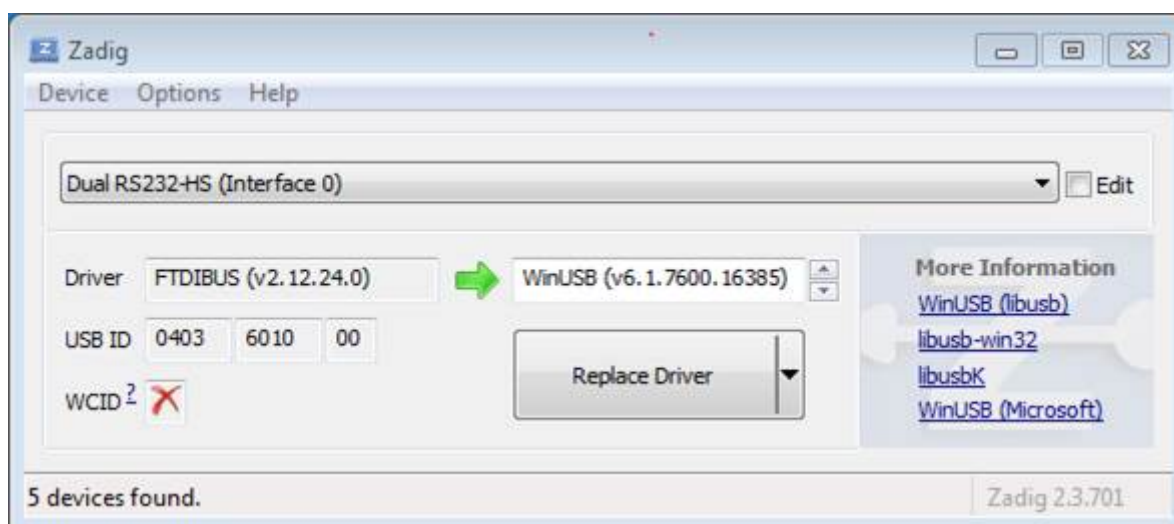


Fig. 37: Configuration of JTAG USB driver in Zadig tool

6. The first device (Dual RS232-HS (Interface 0)) is connected to the JTAG port of the ESP32. Original “FTDIBUS (vxxxx)” driver of this device should be replaced with “WinUSB (v6xxxxx)” . To do so, select “Dual RS232-HS (Interface 0)” and reinstall attached driver to the “WinUSB (v6xxxxx)” , see picture above.

Note: Do not change the second device “Dual RS232-HS (Interface 1)”. It is routed to ESP32’s serial port (UART) used for upload of application to ESP32’s flash.

Now ESP-WROVER-KIT’s JTAG interface should be available to the OpenOCD. To carry on with debugging environment setup, proceed to section [Run OpenOCD](#).

Linux

1. Using standard USB A / micro USB B cable connect ESP-WROVER-KIT board to the computer. Power on the board.
2. Open a terminal, enter `ls -l /dev/ttyUSB*` command and check, if board’s USB ports are recognized by the OS. You are looking for similar result:

```
user-name@computer-name:~/esp$ ls -l /dev/ttyUSB*
crw-rw---- 1 root dialout 188, 0 Jul 10 19:04 /dev/ttyUSB0
crw-rw---- 1 root dialout 188, 1 Jul 10 19:04 /dev/ttyUSB1
```

3. Following section “Permissions delegation” in [OpenOCD’s README](#), set up the access permissions to both USB ports.
4. Log off and login, then cycle the power to the board to make the changes effective. In terminal enter again `ls -l /dev/ttyUSB*` command to verify, if group-owner has changed from dialout to plugdev:

```
user-name@computer-name:~/esp$ ls -l /dev/ttyUSB*
crw-rw-r-- 1 root plugdev 188, 0 Jul 10 19:07 /dev/ttyUSB0
crw-rw-r-- 1 root plugdev 188, 1 Jul 10 19:07 /dev/ttyUSB1
```

If you see similar result and you are a member of plugdev group, then the set up is complete.

The `/dev/ttyUSBn` interface with lower number is used for JTAG communication. The other interface is routed to ESP32’s serial port (UART) used for upload of application to ESP32’s flash.

Now ESP-WROVER-KIT’s JTAG interface should be available to the OpenOCD. To carry on with debugging environment setup, proceed to section [Run OpenOCD](#).

MacOS On macOS, using FT232 for JTAG and serial port at the same time needs some additional steps. When the OS loads FTDI serial port driver, it does so for both channels of FT232 chip. However only one of these channels is used as a serial port, while the other is used as JTAG. If the OS has loaded FTDI serial port driver for the channel used for JTAG, OpenOCD will not be able to connect to the chip. There are two ways around this:

1. Manually unload the FTDI serial port driver before starting OpenOCD, start OpenOCD, then load the serial port driver.
2. Modify FTDI driver configuration so that it doesn’t load itself for channel B of FT232 chip, which is the channel used for JTAG on WROVER KIT.

Manually unloading the driver

1. Install FTDI driver from <http://www.ftdichip.com/Drivers/VCP.htm>
2. Connect USB cable to the WROVER KIT.
3. Unload the serial port driver:

```
sudo kextunload -b com.FTDI.driver.FTDIUSBSerialDriver
```

In some cases you may need to unload Apple’s FTDI driver as well:

```
sudo kextunload -b com.apple.driver.AppleUSBFTDI
```

4. Run OpenOCD:

```
bin/openocd -f board/esp32-wrover-kit-3.3v.cfg
```

5. In another terminal window, load FTDI serial port driver again:

```
sudo kextload -b com.FTDI.driver.FTDIUSBSerialDriver
```

Note: If you need to restart OpenOCD, there is no need to unload FTDI driver again — just stop OpenOCD and start it again. The driver only needs to be unloaded if WROVER KIT was reconnected or power was toggled.

This procedure can be wrapped into a shell script, if desired.

Modifying FTDI driver In a nutshell, this approach requires modification to FTDI driver configuration file, which prevents the driver from being loaded for channel B of FT2232H.

Note: Other boards may use channel A for JTAG, so use this option with caution.

Warning: This approach also needs signature verification of drivers to be disabled, so may not be acceptable for all users.

1. Open FTDI driver configuration file using a text editor (note `sudo`):

```
sudo nano /Library/Extensions/FTDIUSBSerialDriver.kext/Contents/Info.plist
```

2. Find and delete the following lines:

```
<key>FT2232H_B</key>
<dict>
  <key>CFBundleIdentifier</key>
  <string>com.FTDI.driver.FTDIUSBSerialDriver</string>
  <key>IOClass</key>
  <string>FTDIUSBSerialDriver</string>
  <key>IOProviderClass</key>
  <string>IOUSBInterface</string>
  <key>bConfigurationValue</key>
  <integer>1</integer>
  <key>bInterfaceNumber</key>
  <integer>1</integer>
  <key>bcdDevice</key>
  <integer>1792</integer>
  <key>idProduct</key>
  <integer>24592</integer>
  <key>idVendor</key>
  <integer>1027</integer>
</dict>
```

3. Save and close the file
4. Disable driver signature verification:
 1. Open Apple logo menu, choose “Restart...”
 2. When you hear the chime after reboot, press CMD+R immediately
 3. Once Recovery mode starts up, open Terminal
 4. Run the command:

```
csrutil enable --without kext
```

5. Restart again

After these steps, serial port and JTAG can be used at the same time.

To carry on with debugging environment setup, proceed to section [Run OpenOCD](#).

Configure Other JTAG Interface Refer to section [Selecting JTAG Adapter](#) for guidance what JTAG interface to select, so it is able to operate with OpenOCD and ESP32. Then follow three configuration steps below to get it working.

Configure Hardware

1. Identify all pins / signals on JTAG interface and ESP32 board, that should be connected to establish communication.

| | ESP32 Pin | JTAG Signal |
|---|---------------|-------------|
| 1 | CHIP_PU | TRST_N |
| 2 | MTDO / GPIO15 | TDO |
| 3 | MTDI / GPIO12 | TDI |
| 4 | MTCK / GPIO13 | TCK |
| 5 | MTMS / GPIO14 | TMS |
| 6 | GND | GND |

2. Verify if ESP32 pins used for JTAG communication are not connected to some other h/w that may disturb JTAG operation.
3. Connect identified pin / signals of ESP32 and JTAG interface.

Configure Drivers You may need to install driver s/w to make JTAG work with computer. Refer to documentation of JTAG adapter, that should provide related details.

Connect Connect JTAG interface to the computer. Power on ESP32 and JTAG interface boards. Check if JTAG interface is visible by computer.

To carry on with debugging environment setup, proceed to section [Run OpenOCD](#).

Run OpenOCD

Once target is configured and connected to computer, you are ready to launch OpenOCD.

Open a terminal and set it up for using the ESP-IDF as described in the [setting up the environment](#) section of the Getting Started Guide. Then run OpenOCD (this command works on Windows, Linux, and macOS):

```
openocd -f board/esp32-wrover-kit-3.3v.cfg
```

Note: The files provided after `-f` above are specific for ESP-WROVER-KIT with [ESP32-WROOM-32](#) module. You may need to provide different files depending on used hardware. For guidance see [Configuration of OpenOCD for specific target](#).

You should now see similar output (this log is for ESP-WROVER-KIT):

```
user-name@computer-name:~/esp/esp-idf$ openocd -f board/esp32-wrover-kit-3.3v.cfg
Open On-Chip Debugger v0.10.0-esp32-20190708 (2019-07-08-11:04)
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.org/doc/doxygen/bugs.html
none separate
adapter speed: 20000 kHz
force hard breakpoints
Info : ftdi: if you experience problems at higher adapter clocks, try the command
↳ "ftdi_tdo_sample_edge falling"
Info : clock speed 20000 kHz
Info : JTAG tap: esp32.cpu0 tap/device found: 0x120034e5 (mfg: 0x272 (Tensilica),
↳ part: 0x2003, ver: 0x1)
```

(continues on next page)

(continued from previous page)

```
Info : JTAG tap: esp32.cpu1 tap/device found: 0x120034e5 (mfg: 0x272 (Tensilica),
↳part: 0x2003, ver: 0x1)
Info : esp32: Debug controller was reset (pwrstat=0x5F, after clear 0x0F).
Info : esp32: Core was reset (pwrstat=0x5F, after clear 0x0F).
```

- If there is an error indicating permission problems, please see the “Permissions delegation” bit in the OpenOCD README file in `~/esp/openocd-esp32` directory.
- In case there is an error finding configuration files, e.g. Can't find `board/esp32-wrover-kit-3.3v.cfg`, check the path after `-s`. This path is used by OpenOCD to look for the files specified after `-f`. Also check if the file is indeed under provided path.
- If you see JTAG errors (`...all ones/...all zeroes`) please check your connections, whether no other signals are connected to JTAG besides ESP32's pins, and see if everything is powered on.

Upload application for debugging

Build and upload your application to ESP32 as usual, see [Step 8. Build the Project](#).

Another option is to write application image to flash using OpenOCD via JTAG with commands like this:

```
openocd -f board/esp32-wrover-kit-3.3v.cfg -c "program_esp filename.bin 0x10000
↳verify exit"
```

OpenOCD flashing command `program_esp` has the following format:

```
program_esp <image_file> <offset> [verify] [reset] [exit]
```

- `image_file` - Path to program image file.
- `offset` - Offset in flash bank to write image.
- `verify` - Optional. Verify flash contents after writing.
- `reset` - Optional. Reset target after programing.
- `exit` - Optional. Finally exit OpenOCD.

You are now ready to start application debugging. Follow steps described in section below.

5.19.6 Launching Debugger

The toolchain for ESP32 features GNU Debugger, in short GDB. It is available with other toolchain programs under filename `xtensa-esp32-elf-gdb`. GDB can be called and operated directly from command line in a terminal. Another option is to call it from within IDE (like Eclipse, Visual Studio Code, etc.) and operate indirectly with help of GUI instead of typing commands in a terminal.

Both options of using debugger are discussed under links below.

- [Eclipse](#)
- [Command Line](#)

It is recommended to first check if debugger works from [Command Line](#) and then move to using [Eclipse](#).

5.19.7 Debugging Examples

This section is intended for users not familiar with GDB. It presents example debugging session from [Eclipse](#) using simple application available under [get-started/blink](#) and covers the following debugging actions:

1. [Navigating through the code, call stack and threads](#)
2. [Setting and clearing breakpoints](#)
3. [Halting the target manually](#)
4. [Stepping through the code](#)
5. [Checking and setting memory](#)
6. [Watching and setting program variables](#)

7. *Setting conditional breakpoints*

Similar debugging actions are provided using GDB from *Command Line*.

Before proceeding to examples, set up your ESP32 target and load it with [get-started/blink](#).

5.19.8 Building OpenOCD from Sources

Please refer to separate documents listed below, that describe build process.

Building OpenOCD from Sources for Windows

The following instructions are alternative to downloading binary OpenOCD from [Espressif GitHub](#). To quickly setup the binary OpenOCD, instead of compiling it yourself, backup and proceed to section *Setup of OpenOCD*.

Note: Following instructions are assumed to be runned in MSYS2 environment with MINGW32 subsystem!

Install Dependencies Install packages that are required to compile OpenOCD:

```
pacman -S --noconfirm --needed autoconf automake git make \  
mingw-w64-i686-gcc \  
mingw-w64-i686-toolchain \  
mingw-w64-i686-libtool \  
mingw-w64-i686-pkg-config \  
mingw-w64-cross-winpthreads-git \  
p7zip
```

Download Sources of OpenOCD The sources for the ESP32-enabled variant of OpenOCD are available from Espressif GitHub under <https://github.com/espressif/openocd-esp32>. To download the sources, use the following commands:

```
cd ~/esp  
git clone --recursive https://github.com/espressif/openocd-esp32.git
```

The clone of sources should be now saved in `~/esp/openocd-esp32` directory.

Downloading libusb Build and export variables for a following OpenOCD compilation:

```
wget https://github.com/libusb/libusb/releases/download/v1.0.22/libusb-1.0.22.7z  
7z x -olibusb ./libusb-1.0.22.7z  
export CPPFLAGS="$CPPFLAGS -I${PWD}/libusb/include/libusb-1.0"  
export LDFLAGS="$LDFLAGS -L${PWD}/libusb/MinGW32/.libs/dll"
```

Build OpenOCD Proceed with configuring and building OpenOCD:

```
cd ~/esp/openocd-esp32  
export CPPFLAGS="$CPPFLAGS -D__USE_MINGW_ANSI_STDIO=1 -Wno-error"; export CFLAGS="  
↪$CFLAGS -Wno-error"  
./bootstrap  
./configure --disable-doxygen-pdf --enable-ftdi --enable-jlink --enable-ulink --  
↪build=i686-w64-mingw32 --host=i686-w64-mingw32  
make  
cp ../libusb/MinGW32/dll/libusb-1.0.dll ./src  
cp /opt/i686-w64-mingw32/bin/libwinpthread-1.dll ./src
```

Optionally you can add `make install` step at the end. Skip it, if you have an existing OpenOCD (from e.g. another development platform), as it may get overwritten. Also you could use `export DESTDIR="/custom/install/dir"; make install`.

Note:

- Should an error occur, resolve it and try again until the command `make` works.
 - If there is a submodule problem from OpenOCD, please `cd` to the `openocd-esp32` directory and input `git submodule update --init`.
 - If the `./configure` is successfully run, information of enabled JTAG will be printed under OpenOCD configuration summary.
 - If the information of your device is not shown in the log, use `./configure` to enable it as described in `../openocd-esp32/doc/INSTALL.txt`.
 - For details concerning compiling OpenOCD, please refer to `openocd-esp32/README.Windows`.
 - Don't forget to copy `libusb-1.0.dll` and `libwinpthread-1.dll` into `OOCD_INSTALLDIR/bin` from `~/esp/openocd-esp32/src`.
-

Once `make` process is successfully completed, the executable of OpenOCD will be saved in `~/esp/openocd-esp32/src` directory.

Full Listing A complete described previously process is provided below for the faster execution, e.g. as a shell script:

```
pacman -S --noconfirm --needed autoconf automake git make mingw-w64-i686-gcc mingw-
↳w64-i686-toolchain mingw-w64-i686-libtool mingw-w64-i686-pkg-config mingw-w64-
↳cross-winpthreads-git p7zip
cd ~/esp
git clone --recursive https://github.com/espressif/openocd-esp32.git

wget https://github.com/libusb/libusb/releases/download/v1.0.22/libusb-1.0.22.7z
7z x -olibusb ./libusb-1.0.22.7z
export CPPFLAGS="$CPPFLAGS -I${PWD}/libusb/include/libusb-1.0"; export LDFLAGS="
↳$LDFLAGS -L${PWD}/libusb/MinGW32/.libs/dll"

export CPPFLAGS="$CPPFLAGS -D__USE_MINGW_ANSI_STDIO=1 -Wno-error"; export CFLAGS="
↳$CFLAGS -Wno-error"
cd ~/esp/openocd-esp32
./bootstrap
./configure --disable-doxxygen-pdf --enable-ftdi --enable-jlink --enable-ulink --
↳build=i686-w64-mingw32 --host=i686-w64-mingw32
make
cp ../libusb/MinGW32/dll/libusb-1.0.dll ./src
cp /opt/i686-w64-mingw32/bin/libwinpthread-1.dll ./src

# # optional
# export DESTDIR="$PWD"
# make install
# cp ./src/libusb-1.0.dll $DESTDIR/mingw32/bin
# cp ./src/libwinpthread-1.dll $DESTDIR/mingw32/bin
```

Next Steps To carry on with debugging environment setup, proceed to section [Configuring ESP32 Target](#).

Building OpenOCD from Sources for Linux

The following instructions are alternative to downloading binary OpenOCD from [Espressif GitHub](#). To quickly setup the binary OpenOCD, instead of compiling it yourself, backup and proceed to section [Setup of OpenOCD](#).

Download Sources of OpenOCD The sources for the ESP32-enabled variant of OpenOCD are available from Espressif GitHub under <https://github.com/espressif/openocd-esp32>. To download the sources, use the following commands:

```
cd ~/esp
git clone --recursive https://github.com/espressif/openocd-esp32.git
```

The clone of sources should be now saved in `~/esp/openocd-esp32` directory.

Install Dependencies Install packages that are required to compile OpenOCD.

Note: Install the following packages one by one, check if installation was successful and then proceed to the next package. Resolve reported problems before moving to the next step.

```
sudo apt-get install make
sudo apt-get install libtool
sudo apt-get install pkg-config
sudo apt-get install autoconf
sudo apt-get install automake
sudo apt-get install texinfo
sudo apt-get install libusb-1.0
```

Note:

- Version of `pkg-config` should be 0.2.3 or above.
 - Version of `autoconf` should be 2.6.4 or above.
 - Version of `automake` should be 1.9 or above.
 - When using USB-Blaster, ASIX Presto, OpenJTAG and FT2232 as adapters, drivers `libFTDI` and `FTD2XX` need to be downloaded and installed.
 - When using CMSIS-DAP, `HIDAPI` is needed.
-

Build OpenOCD Proceed with configuring and building OpenOCD:

```
cd ~/esp/openocd-esp32
./bootstrap
./configure
make
```

Optionally you can add `sudo make install` step at the end. Skip it, if you have an existing OpenOCD (from e.g. another development platform), as it may get overwritten.

Note:

- Should an error occur, resolve it and try again until the command `make` works.
 - If there is a submodule problem from OpenOCD, please `cd` to the `openocd-esp32` directory and input `git submodule update --init`.
 - If the `./configure` is successfully run, information of enabled JTAG will be printed under OpenOCD configuration summary.
 - If the information of your device is not shown in the log, use `./configure` to enable it as described in `../openocd-esp32/doc/INSTALL.txt`.
 - For details concerning compiling OpenOCD, please refer to `openocd-esp32/README`.
-

Once `make` process is successfully completed, the executable of OpenOCD will be saved in `~/openocd-esp32/bin` directory.

Next Steps To carry on with debugging environment setup, proceed to section [Configuring ESP32 Target](#).

Building OpenOCD from Sources for MacOS

The following instructions are alternative to downloading binary OpenOCD from [Espressif GitHub](#). To quickly setup the binary OpenOCD, instead of compiling it yourself, backup and proceed to section [Setup of OpenOCD](#).

Download Sources of OpenOCD The sources for the ESP32-enabled variant of OpenOCD are available from Espressif GitHub under <https://github.com/espressif/openocd-esp32>. To download the sources, use the following commands:

```
cd ~/esp
git clone --recursive https://github.com/espressif/openocd-esp32.git
```

The clone of sources should be now saved in `~/esp/openocd-esp32` directory.

Install Dependencies Install packages that are required to compile OpenOCD using Homebrew:

```
brew install automake libtool libusb wget gcc@4.9 pkg-config
```

Build OpenOCD Proceed with configuring and building OpenOCD:

```
cd ~/esp/openocd-esp32
./bootstrap
./configure
make
```

Optionally you can add `sudo make install` step at the end. Skip it, if you have an existing OpenOCD (from e.g. another development platform), as it may get overwritten.

Note:

- Should an error occur, resolve it and try again until the command `make` works.
- If there is a submodule problem from OpenOCD, please `cd` to the `openocd-esp32` directory and input `git submodule update --init`.
- If the `./configure` is successfully run, information of enabled JTAG will be printed under OpenOCD configuration summary.
- If the information of your device is not shown in the log, use `./configure` to enable it as described in `../openocd-esp32/doc/INSTALL.txt`.
- For details concerning compiling OpenOCD, please refer to `openocd-esp32/README.OSX`.

Once `make` process is successfully completed, the executable of OpenOCD will be saved in `~/esp/openocd-esp32/src/openocd` directory.

Next Steps To carry on with debugging environment setup, proceed to section [Configuring ESP32 Target](#).

The examples of invoking OpenOCD in this document assume using pre-built binary distribution described in section [Setup of OpenOCD](#).

To use binaries build locally from sources, change the path to OpenOCD executable to `src/openocd` and set the `OPENOCD_SCRIPTS` environment variable so that OpenOCD can find the configuration files. For Linux and macOS:

```
cd ~/esp/openocd-esp32
export OPENOCD_SCRIPTS=$PWD/tcl
```

For Windows:

```
cd %USERPROFILE%\esp\openocd-esp32
set "OPENOCD_SCRIPTS=%CD%\tcl"
```

Example of invoking OpenOCD build locally from sources, for Linux and macOS:

```
src/openocd -f board/esp32-wrover-kit-3.3v.cfg
```

and Windows:

```
src\openocd -f board\esp32-wrover-kit-3.3v.cfg
```

5.19.9 Tips and Quirks

This section provides collection of links to all tips and quirks referred to from various parts of this guide.

- [Breakpoints and watchpoints available](#)
- [What else should I know about breakpoints?](#)
- [Flash Mappings vs SW Flash Breakpoints](#)
- [Why stepping with “next” does not bypass subroutine calls?](#)
- [Support options for OpenOCD at compile time](#)
- [FreeRTOS support](#)
- [Why to set SPI flash voltage in OpenOCD configuration?](#)
- [Optimize JTAG speed](#)
- [What is the meaning of debugger’s startup commands?](#)
- [Configuration of OpenOCD for specific target](#)
- [How debugger resets ESP32?](#)
- [Do not use JTAG pins for something else](#)
- [JTAG and ESP32-WROOM-32 AT firmware Compatibility Issue](#)
- [Reporting issues with OpenOCD / GDB](#)

5.19.10 Related Documents

Using Debugger

This section covers configuration and running debugger either from [Eclipse](#) or [Command Line](#). It is recommended to first check if debugger works from [Command Line](#) and then move to using Eclipse.

Eclipse Debugging functionality is provided out of box in standard Eclipse installation. Another option is to use pluggins like “GDB Hardware Debugging” plugin. We have found this plugin quite convenient and decided to use throughout this guide.

To begin with, install “GDB Hardware Debugging” plugin by opening Eclipse and going to *Help > Install New Software*.

Once installation is complete, configure debugging session following steps below. Please note that some of configuration parameters are generic and some are project specific. This will be shown below by configuring debugging for “blink” example project. If not done already, add this project to Eclipse workspace following guidance in section [Build and Flash with Eclipse IDE](#). The source of [get-started/blink](#) application is available in [examples](#) directory of ESP-IDF repository.

1. In Eclipse go to *Run > Debug Configuration*. A new window will open. In the window’s left pane double click “GDB Hardware Debugging” (or select “GDB Hardware Debugging” and press the “New” button) to create a new configuration.
2. In a form that will show up on the right, enter the “Name:” of this configuration, e.g. “Blink checking” .
3. On the “Main” tab below, under “Project:” , press “Browse” button and select the “blink” project.

4. In next line “C/C++ Application:” press “Browse” button and select “blink.elf” file. If “blink.elf” is not there, then likely this project has not been build yet. See [Build and Flash with Eclipse IDE](#) how to do it.
5. Finally, under “Build (if required) before launching” click “Disable auto build” .
A sample window with settings entered in points 1 - 5 is shown below.

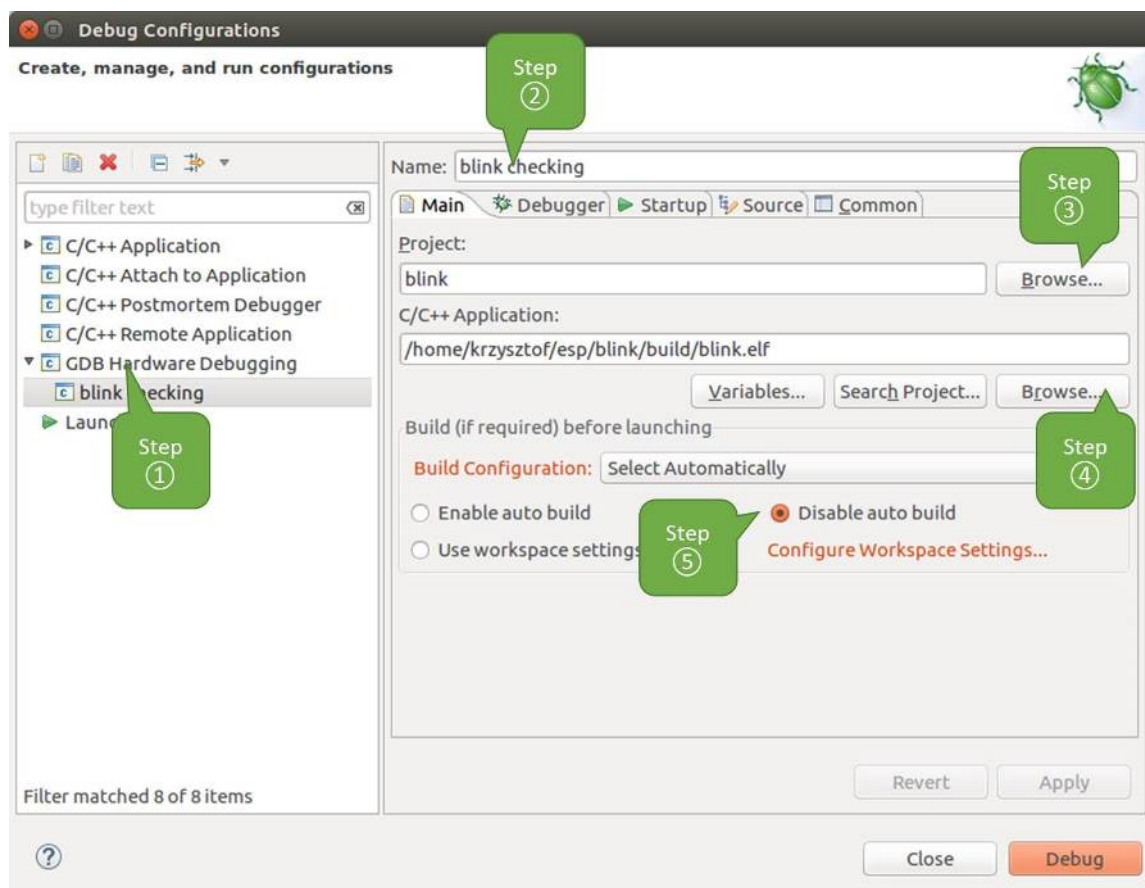


Fig. 38: Configuration of GDB Hardware Debugging - Main tab

6. Click “Debugger” tab. In field “GDB Command” enter `xtensa-esp32-elf-gdb` to invoke debugger.
7. Change default configuration of “Remote host” by entering 3333 under the “Port number” .
Configuration entered in points 6 and 7 is shown on the following picture.
8. The last tab to that requires changing of default configuration is “Startup” . Under “Initialization Commands” uncheck “Reset and Delay (seconds)” and “Halt” . Then, in entry field below, enter the following lines:

```
mon reset halt
flushregs
set remote hardware-watchpoint-limit 2
```

Note: If you want to update image in the flash automatically before starting new debug session add the following lines of commands at the beginning of “Initialization Commands” textbox:

```
mon reset halt
mon program_esp ${workspace_loc:blink/build/blink.bin} 0x10000 verify
```

For description of `program_esp` command see [Upload application for debugging](#).

9. Under “Load Image and Symbols” uncheck “Load image” option.
10. Further down on the same tab, establish an initial breakpoint to halt CPUs after they are reset by debugger. The plugin will set this breakpoint at the beginning of the function entered under “Set break point at:” . Checkout this option and enter `app_main` in provided field.
11. Checkout “Resume” option. This will make the program to resume after `mon reset halt` is invoked per point 8. The program will then stop at breakpoint inserted at `app_main`.

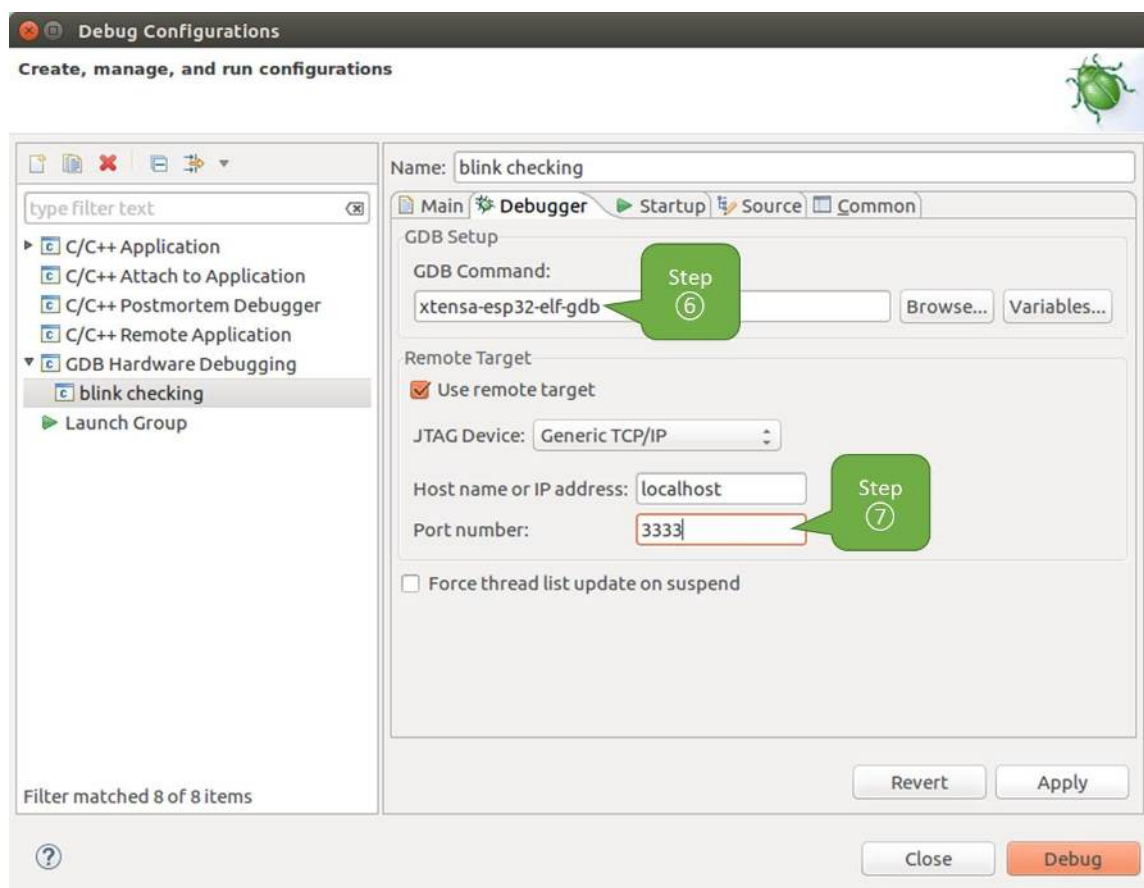


Fig. 39: Configuration of GDB Hardware Debugging - Debugger tab

Configuration described in points 8 - 11 is shown below.

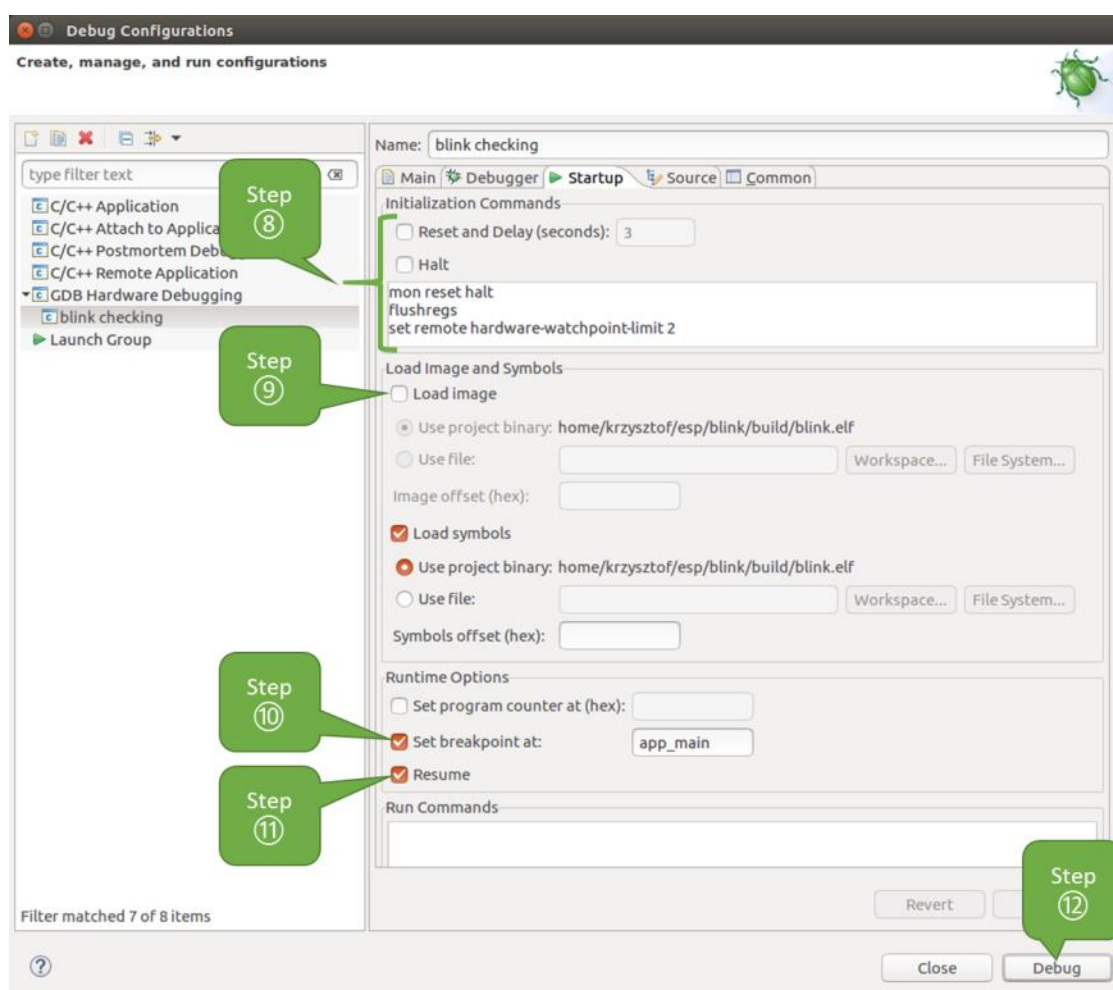


Fig. 40: Configuration of GDB Hardware Debugging - Startup tab

If the “Startup” sequence looks convoluted and respective “Initialization Commands” are not clear to you, check [What is the meaning of debugger’s startup commands?](#) for additional explanation.

12. If you previously completed [Configuring ESP32 Target](#) steps described above, so the target is running and ready to talk to debugger, go right to debugging by pressing “Debug” button. Otherwise press “Apply” to save changes, go back to [Configuring ESP32 Target](#) and return here to start debugging.

Once all 1 - 12 configuration steps are satisfied, the new Eclipse perspective called “Debug” will open as shown on example picture below.

If you are not quite sure how to use GDB, check [Eclipse](#) example debugging session in section [Debugging Examples](#).

Command Line

1. Begin with completing steps described under [Configuring ESP32 Target](#). This is prerequisite to start a debugging session.
2. Open a new terminal session and go to directory that contains project for debugging, e.g.

```
cd ~/esp/blink
```

3. When launching a debugger, you will need to provide couple of configuration parameters and commands. Instead of entering them one by one in command line, create a configuration file and name it `gdbinit`:

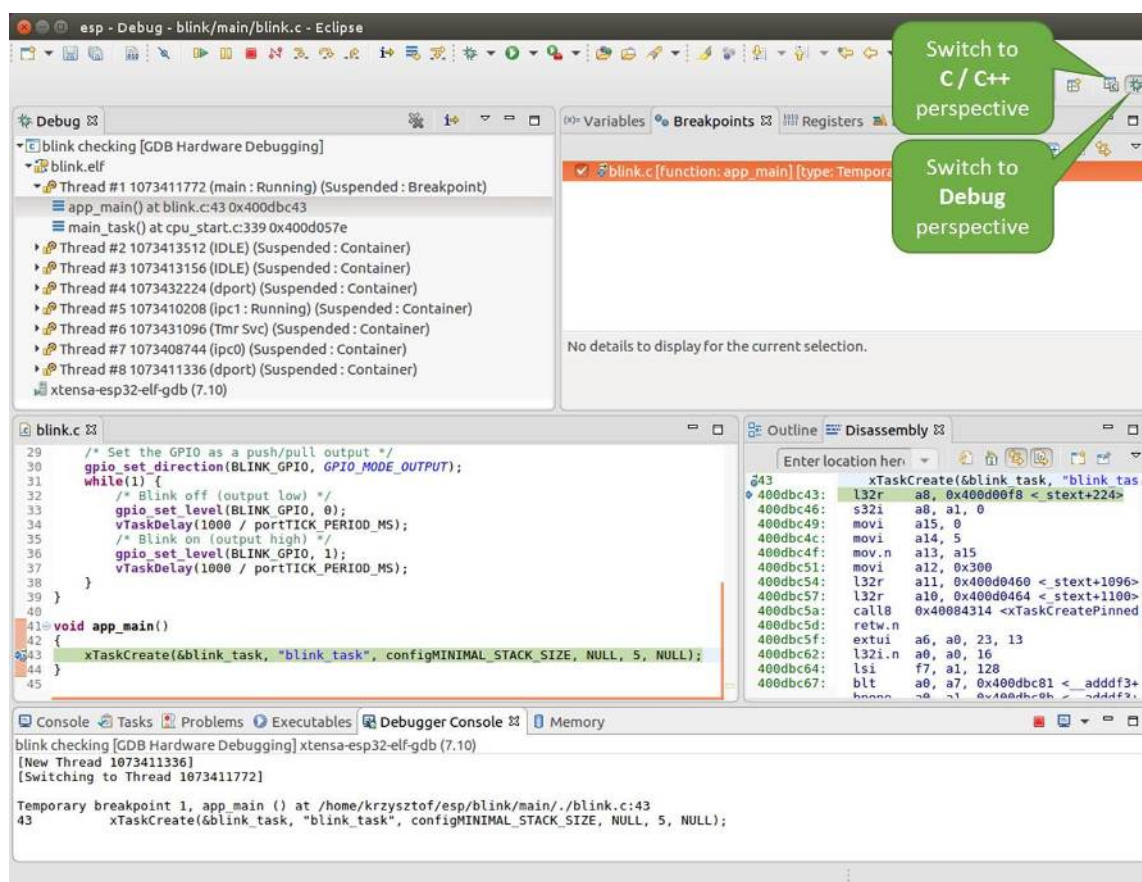


Fig. 41: Debug Perspective in Eclipse


```
target remote :3333
set remote hardware-watchpoint-limit 2
mon reset halt
flushregs
thb app_main
c
```

Save this file in current directory.

For more details what's inside `gdbinit` file, see [What is the meaning of debugger's startup commands?](#)

- Now you are ready to launch GDB. Type the following in terminal:

```
xtensa-esp32-elf-gdb -x gdbinit build/blink.elf
```

- If previous steps have been done correctly, you will see a similar log concluded with `(gdb)` prompt:

```
user-name@computer-name:~/esp/blink$ xtensa-esp32-elf-gdb -x gdbinit build/
↳blink.elf
GNU gdb (crosstool-NG crosstool-ng-1.22.0-61-gab8375a) 7.10
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=x86_64-build_pc-linux-gnu --target=xtensa-
↳esp32-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from build/blink.elf...done.
0x400d10d8 in esp_vApplicationIdleHook () at /home/user-name/esp/esp-idf/
↳components/esp32/./freertos_hooks.c:52
52     asm("waiti 0");
JTAG tap: esp32.cpu0 tap/device found: 0x120034e5 (mfg: 0x272 (Tensilica),
↳part: 0x2003, ver: 0x1)
JTAG tap: esp32.slave tap/device found: 0x120034e5 (mfg: 0x272 (Tensilica),
↳part: 0x2003, ver: 0x1)
esp32: Debug controller was reset (pwrstat=0x5F, after clear 0x0F).
esp32: Core was reset (pwrstat=0x5F, after clear 0x0F).
Target halted. PRO_CPU: PC=0x5000004B (active) APP_CPU: PC=0x00000000
esp32: target state: halted
esp32: Core was reset (pwrstat=0x1F, after clear 0x0F).
Target halted. PRO_CPU: PC=0x40000400 (active) APP_CPU: PC=0x40000400
esp32: target state: halted
Hardware assisted breakpoint 1 at 0x400db717: file /home/user-name/esp/blink/
↳main/./blink.c, line 43.
0x0: 0x00000000
Target halted. PRO_CPU: PC=0x400DB717 (active) APP_CPU: PC=0x400D10D8
[New Thread 1073428656]
[New Thread 1073413708]
[New Thread 1073431316]
[New Thread 1073410672]
[New Thread 1073408876]
[New Thread 1073432196]
[New Thread 1073411552]
[Switching to Thread 1073411996]

Temporary breakpoint 1, app_main () at /home/user-name/esp/blink/main/./blink.
↳c:43
```

(continues on next page)

(continued from previous page)

```

43     xTaskCreate(&blink_task, "blink_task", 512, NULL, 5, NULL);
(gdb)

```

Note the third line from bottom that shows debugger halting at breakpoint established in `gdbinit` file at function `app_main()`. Since the processor is halted, the LED should not be blinking. If this is what you see as well, you are ready to start debugging.

If you are not quite sure how to use GDB, check [Command Line](#) example debugging session in section [Debugging Examples](#).

Debugging Examples

This section describes debugging with GDB from [Eclipse](#) as well as from [Command Line](#).

Eclipse Verify if your target is ready and loaded with [get-started/blink](#) example. Configure and start debugger following steps in section [Eclipse](#). Pick up where target was left by debugger, i.e. having the application halted at breakpoint established at `app_main()`.

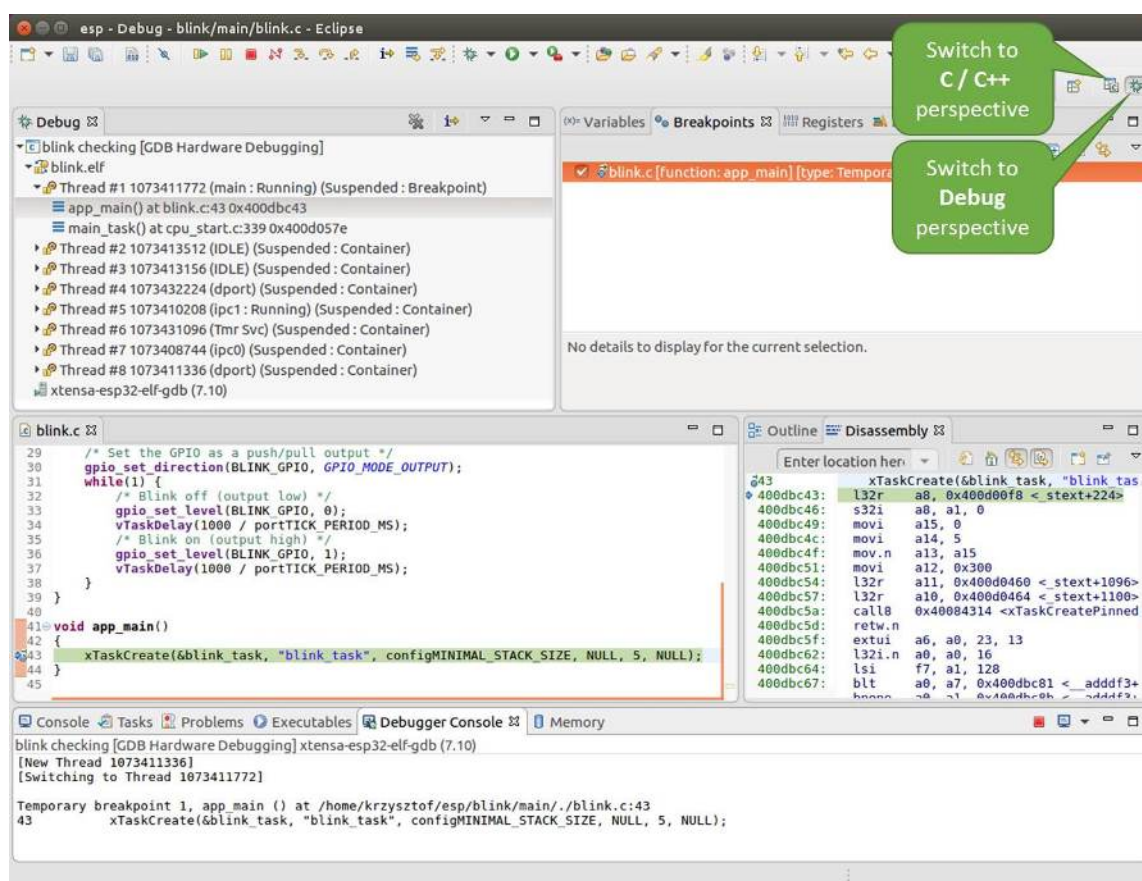


Fig. 42: Debug Perspective in Eclipse

Examples in this section

1. [Navigating through the code, call stack and threads](#)
2. [Setting and clearing breakpoints](#)
3. [Halting the target manually](#)
4. [Stepping through the code](#)
5. [Checking and setting memory](#)

6. *Watching and setting program variables*
7. *Setting conditional breakpoints*

Navigating through the code, call stack and threads When the target is halted, debugger shows the list of threads in “Debug” window. The line of code where program halted is highlighted in another window below, as shown on the following picture. The LED stops blinking.

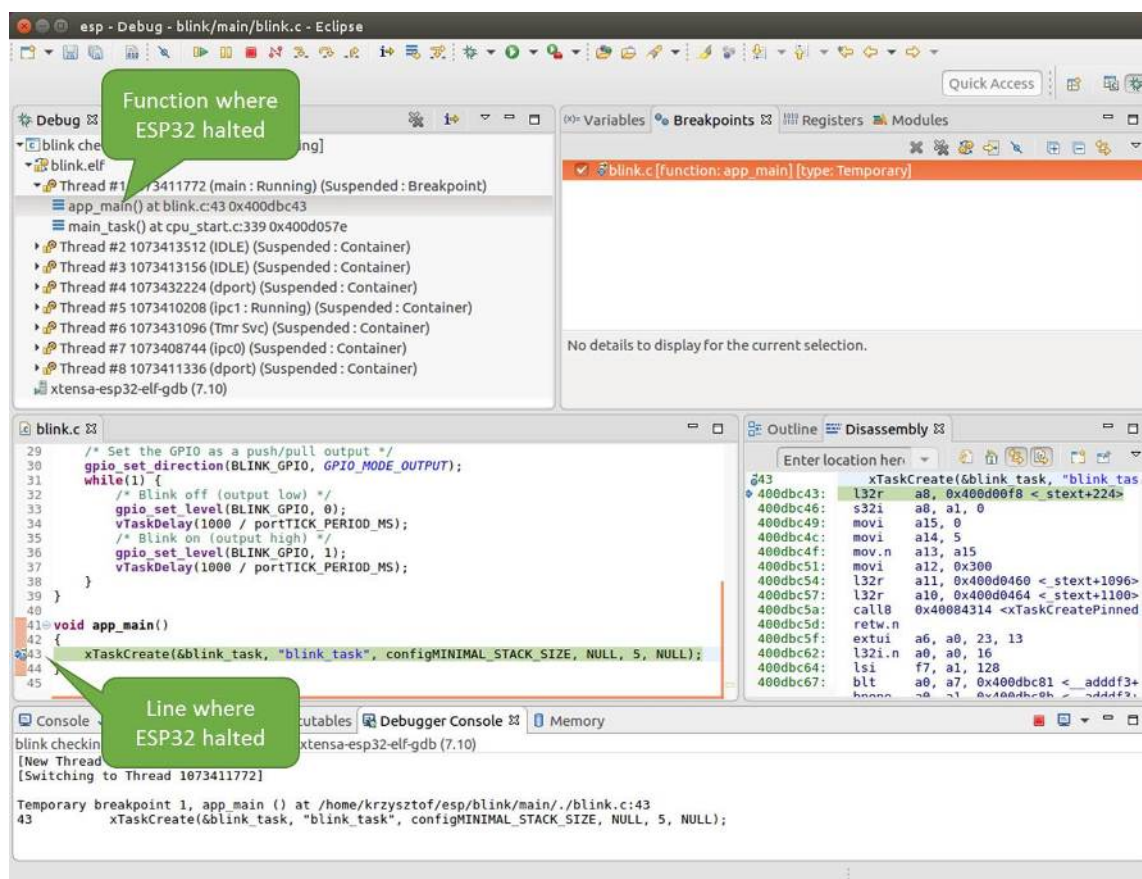


Fig. 43: Target halted during debugging

Specific thread where the program halted is expanded showing the call stack. It represents function calls that lead up to the highlighted line of code, where the target halted. The first line of call stack under Thread #1 contains the last called function `app_main()`, that in turn was called from function `main_task()` shown in a line below. Each line of the stack also contains the file name and line number where the function was called. By clicking / highlighting the stack entries, in window below, you will see contents of this file.

By expanding threads you can navigate throughout the application. Expand Thread #5 that contains much longer call stack. You will see there, besides function calls, numbers like `0x4000000c`. They represent addresses of binary code not provided in source form.

In another window on right, you can see the disassembled machine code no matter if your project provides it in source or only the binary form.

Go back to the `app_main()` in Thread #1 to familiar code of `blink.c` file that will be examined in more details in the following examples. Debugger makes it easy to navigate through the code of entire application. This comes handy when stepping through the code and working with breakpoints and will be discussed below.

Setting and clearing breakpoints When debugging, we would like to be able to stop the application at critical lines of code and then examine the state of specific variables, memory and registers / peripherals. To do so we are using breakpoints. They provide a convenient way to quickly get to and halt the application at specific line.

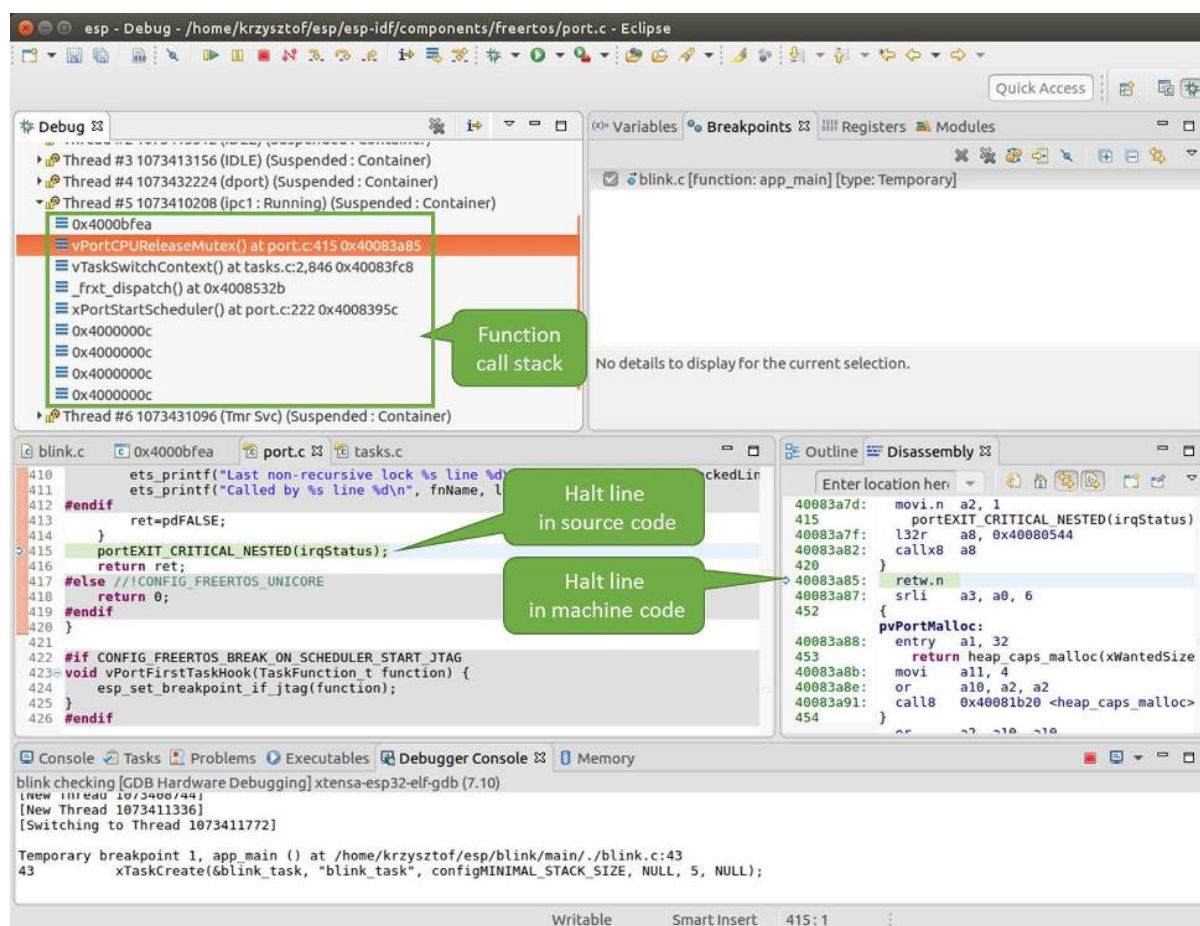


Fig. 44: Navigate through the call stack

Let's establish two breakpoints when the state of LED changes. Basing on code listing above, this happens at lines 33 and 36. To do so, hold the “Control” on the keyboard and double click on number 33 in file `blink.c` file. A dialog will open where you can confirm your selection by pressing “OK” button. If you do not like to see the dialog just double click the line number. Set another breakpoint in line 36.

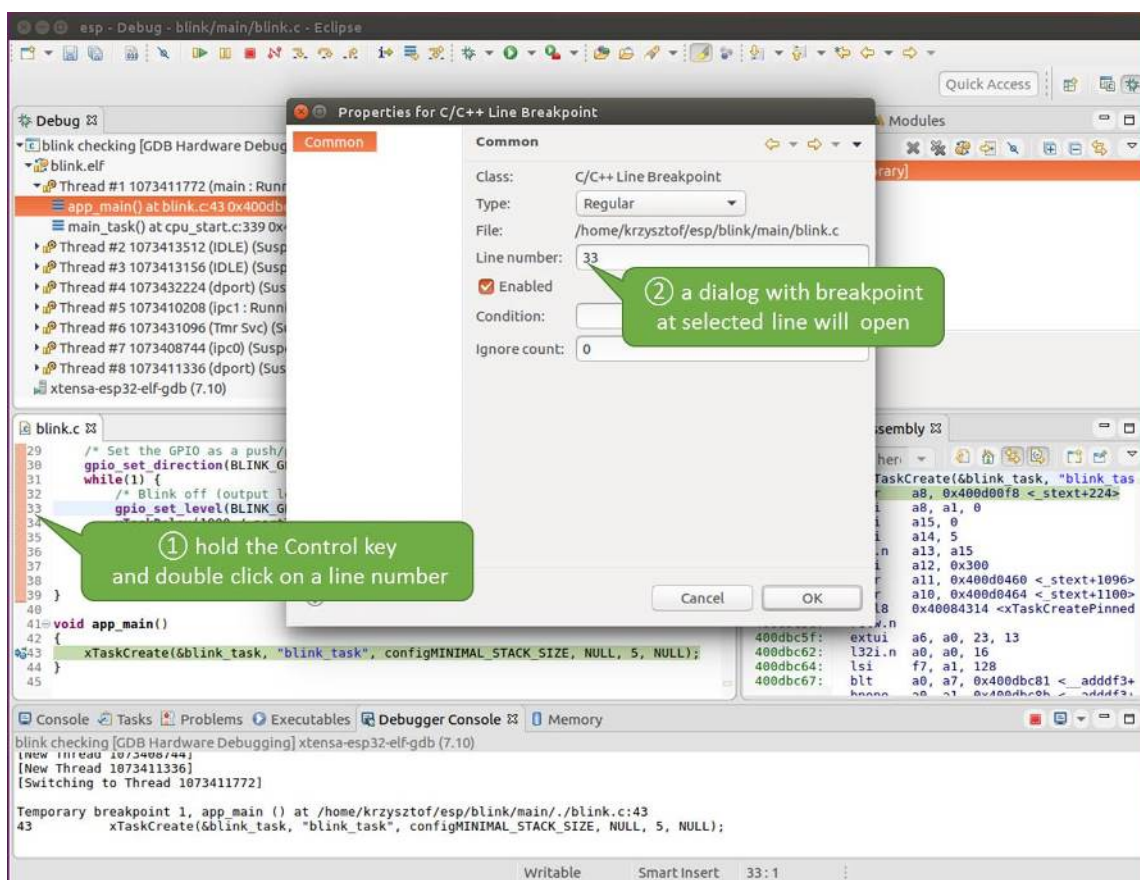


Fig. 45: Setting a breakpoint

Information how many breakpoints are set and where is shown in window “Breakpoints” on top right. Click “Show Breakpoints Supported by Selected Target” to refresh this list. Besides the two just set breakpoints the list may contain temporary breakpoint at function `app_main()` established at debugger start. As maximum two breakpoints are allowed (see [Breakpoints and watchpoints available](#)), you need to delete it, or debugging will fail.

If you now click “Resume” (click `blink_task()` under “Tread #8” , if “Resume” button is grayed out), the processor will run and halt at a breakpoint. Clicking “Resume” another time will make it run again, halt on second breakpoint, and so on.

You will be also able to see that LED is changing the state after each click to “Resume” program execution.

Read more about breakpoints under [Breakpoints and watchpoints available](#) and [What else should I know about breakpoints?](#)

Halting the target manually When debugging, you may resume application and enter code waiting for some event or staying in infinite loop without any break points defined. In such case, to go back to debugging mode, you can break program execution manually by pressing “Suspend” button.

To check it, delete all breakpoints and click “Resume” . Then click “Suspend” . Application will be halted at some random point and LED will stop blinking. Debugger will expand tread and highlight the line of code where application halted.

In particular case above, the application has been halted in line 52 of code in file `freertos_hooks.c` Now you can resume it again by pressing “Resume” button or do some debugging as discussed below.

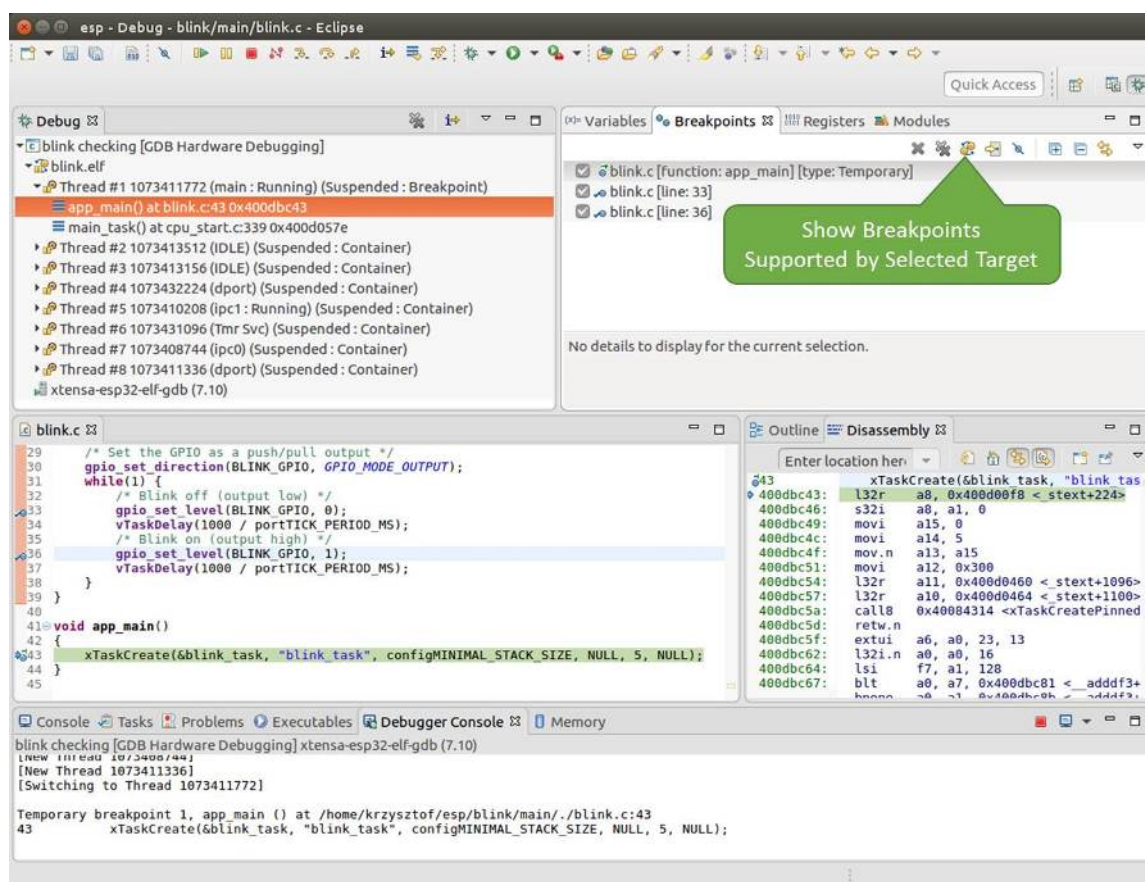


Fig. 46: Three breakpoints are set / maximum two are allowed

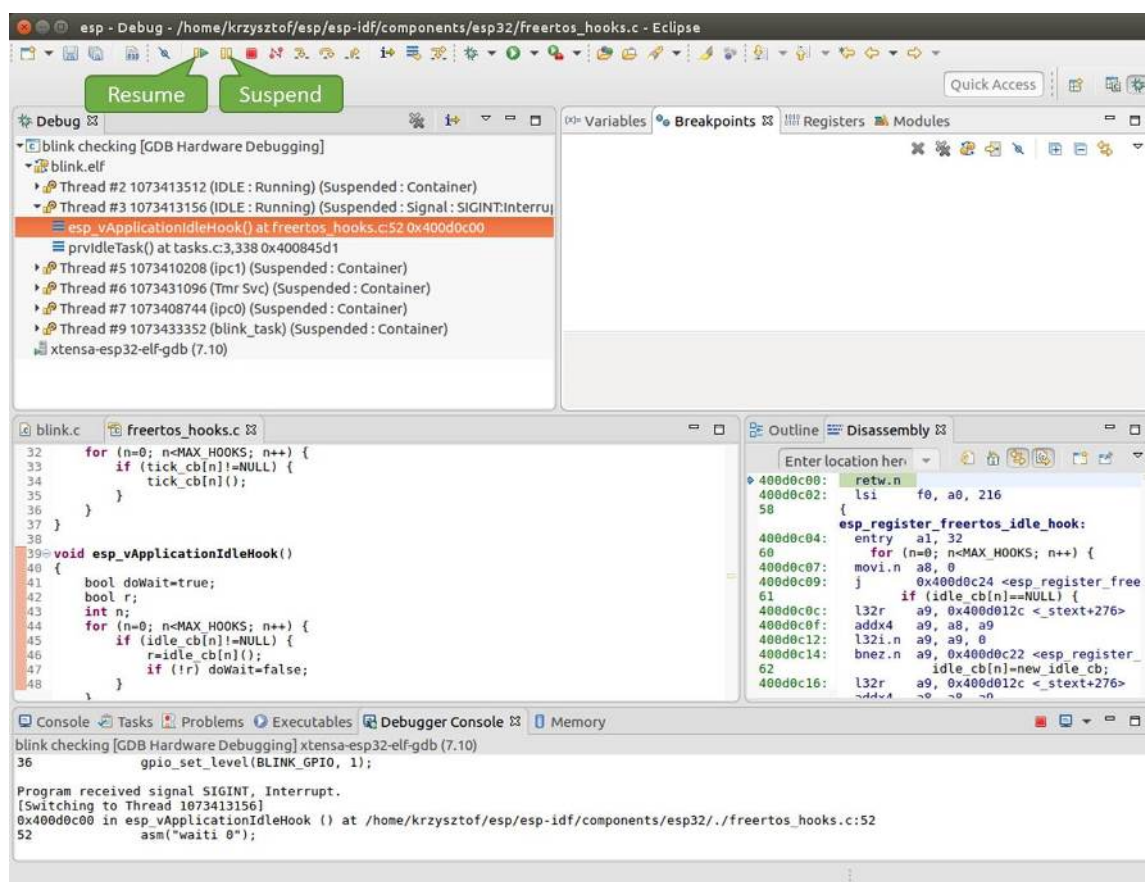


Fig. 47: Target halted manually

Stepping through the code It is also possible to step through the code using “Step Into (F5)” and “Step Over (F6)” commands. The difference is that “Step Into (F5)” is entering inside subroutine calls, while “Step Over (F6)” steps over the call, treating it as a single source line.

Before being able to demonstrate this functionality, using information discussed in previous paragraph, make sure that you have only one breakpoint defined at line 36 of `blink.c`.

Resume program by entering pressing F8 and let it halt. Now press “Step Over (F6)”, one by one couple of times, to see how debugger is stepping one program line at a time.

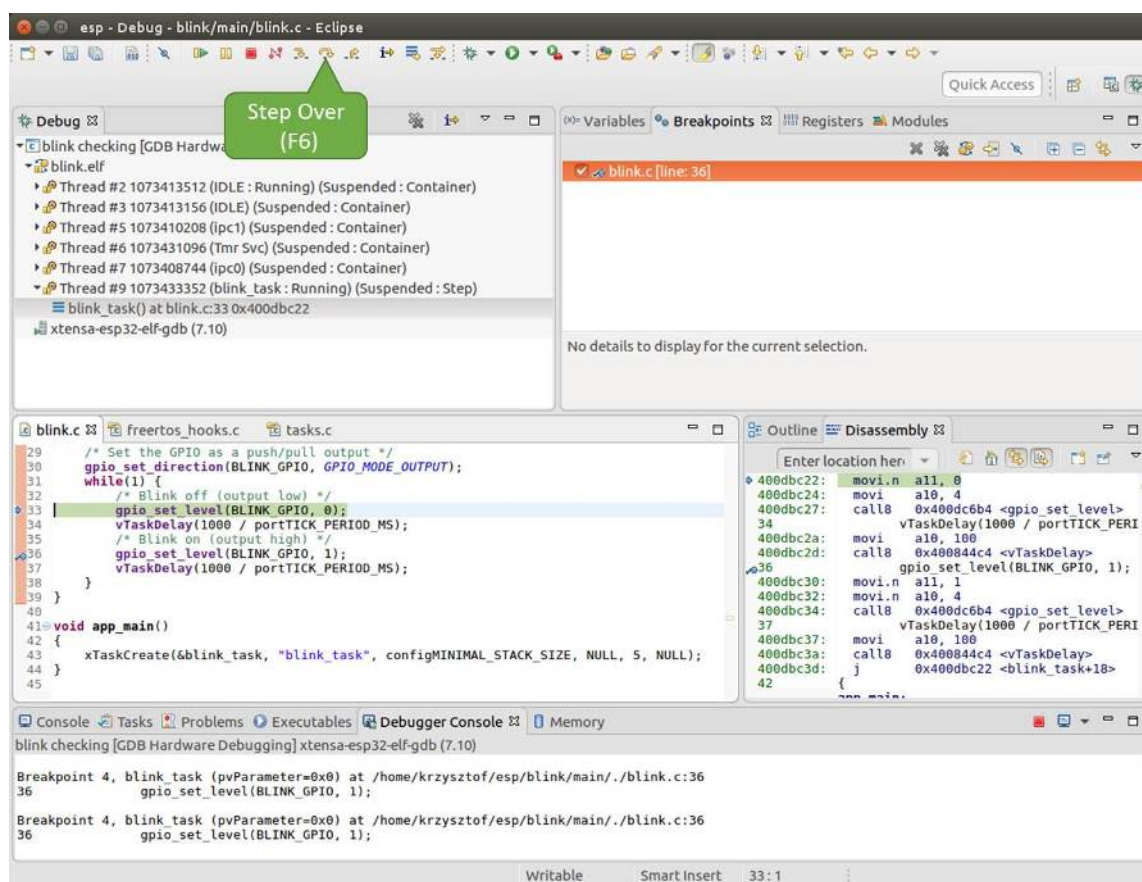


Fig. 48: Stepping through the code with “Step Over (F6)”

If you press “Step Into (F5)” instead, then debugger will step inside subroutine calls.

In this particular case debugger stepped inside `gpio_set_level(BLINK_GPIO, 0)` and effectively moved to `gpio.c` driver code.

See [Why stepping with “next” does not bypass subroutine calls?](#) for potential limitation of using `next` command.

Checking and setting memory To display or set contents of memory use “Memory” tab at the bottom of “Debug” perspective.

With the “Memory” tab, we will read from and write to the memory location `0x3FF44004` labeled as `GPIO_OUT_REG` used to set and clear individual GPIO’s. For more information please refer to [ESP32 Technical Reference Manual](#), chapter `IO_MUX` and `GPIO Matrix`.

Being in the same `blink.c` project as before, set two breakpoints right after `gpio_set_level` instruction. Click “Memory” tab and then “Add Memory Monitor” button. Enter `0x3FF44004` in provided dialog.

Now resume program by pressing F8 and observe “Monitor” tab.

You should see one bit being flipped over at memory location `0x3FF44004` (and LED changing the state) each time F8 is pressed.

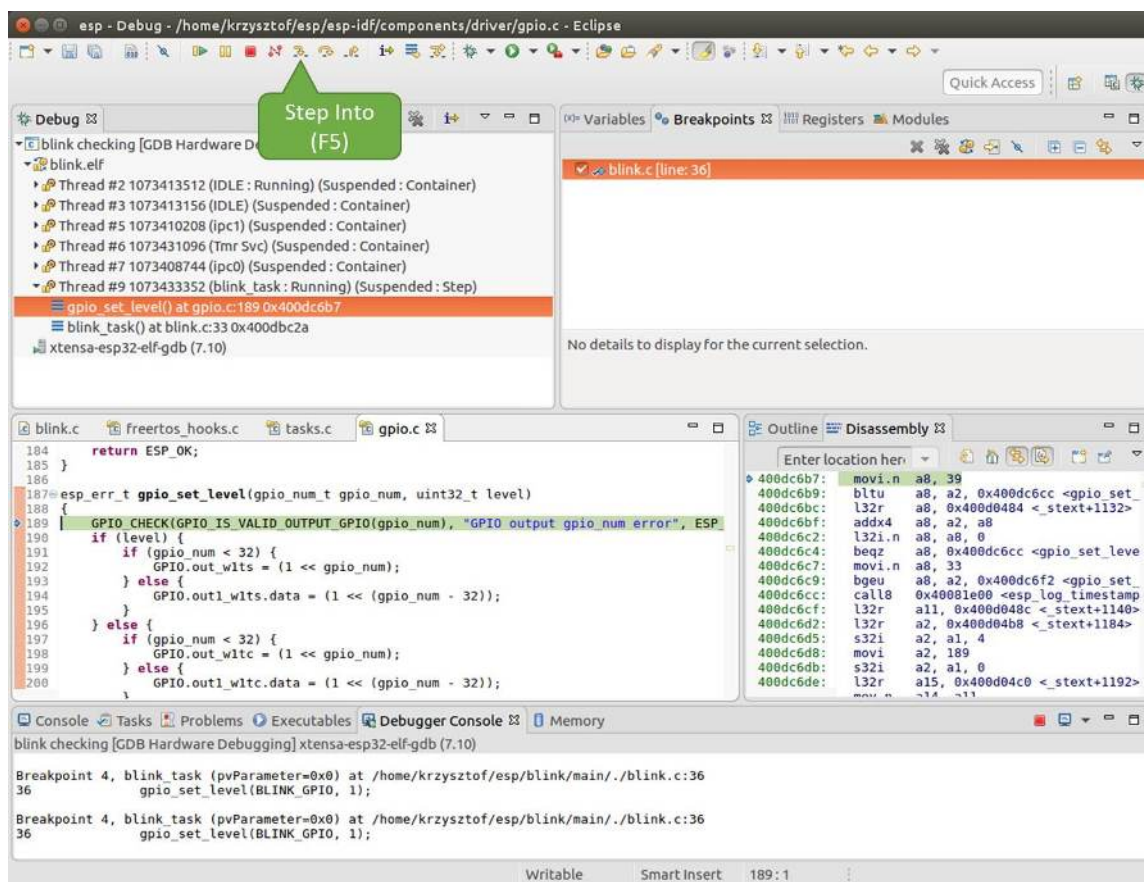


Fig. 49: Stepping through the code with “Step Into (F5)”

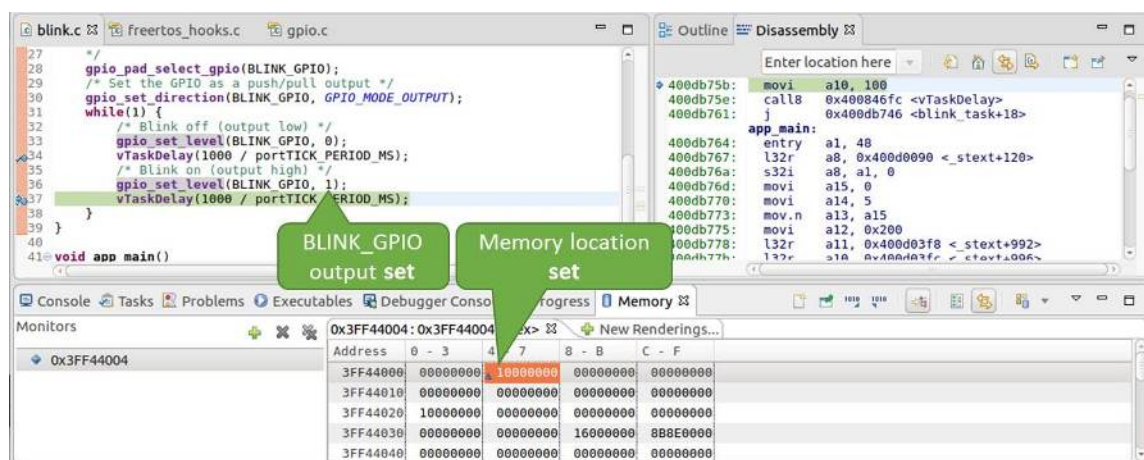


Fig. 50: Observing memory location 0x3FF44004 changing one bit to “ON”

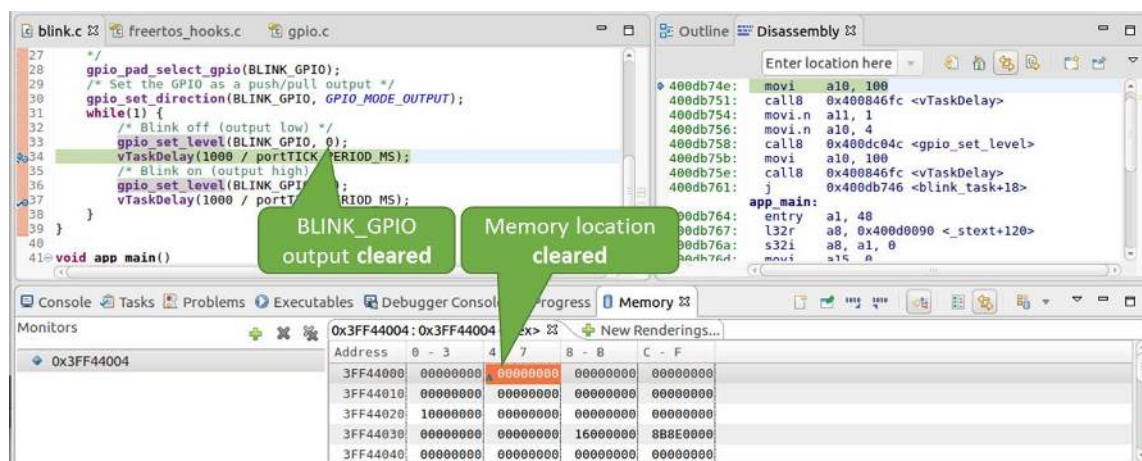


Fig. 51: Observing memory location 0x3FF44004 changing one bit to “OFF”

To set memory use the same “Monitor” tab and the same memory location. Type in alternate bit pattern as previously observed. Immediately after pressing enter you will see LED changing the state.

Watching and setting program variables A common debugging task is checking the value of a program variable as the program runs. To be able to demonstrate this functionality, update file `blink.c` by adding a declaration of a global variable `int i` above definition of function `blink_task`. Then add `i++` inside `while(1)` of this function to get `i` incremented on each blink.

Exit debugger, so it is not confused with new code, build and flash the code to the ESP and restart debugger. There is no need to restart OpenOCD.

Once application is halted, enter a breakpoint in the line where you put `i++`.

In next step, in the window with “Breakpoints”, click the “Expressions” tab. If this tab is not visible, then add it by going to the top menu `Window > Show View > Expressions`. Then click “Add new expression” and enter `i`.

Resume program execution by pressing F8. Each time the program is halted you will see `i` value being incremented.

To modify `i` enter a new number in “Value” column. After pressing “Resume (F8)” the program will keep incrementing `i` starting from the new entered number.

Setting conditional breakpoints Here comes more interesting part. You may set a breakpoint to halt the program execution, if certain condition is satisfied. Right click on the breakpoint to open a context menu and select “Breakpoint Properties”. Change the selection under “Type:” to “Hardware” and enter a “Condition:” like `i == 2`.

If current value of `i` is less than 2 (change it if required) and program is resumed, it will blink LED in a loop until condition `i == 2` gets true and then finally halt.

Command Line Verify if your target is ready and loaded with [get-started/blink](#) example. Configure and start debugger following steps in section [Command Line](#). Pick up where target was left by debugger, i.e. having the application halted at breakpoint established at `app_main()`:

```
Temporary breakpoint 1, app_main () at /home/user-name/esp/blink/main/./blink.c:43
43      xTaskCreate(&blink_task, "blink_task", configMINIMAL_STACK_SIZE, NULL, 5, &
↳NULL);
(gdb)
```

Examples in this section

1. [Navigating through the code, call stack and threads](#)
2. [Setting and clearing breakpoints](#)

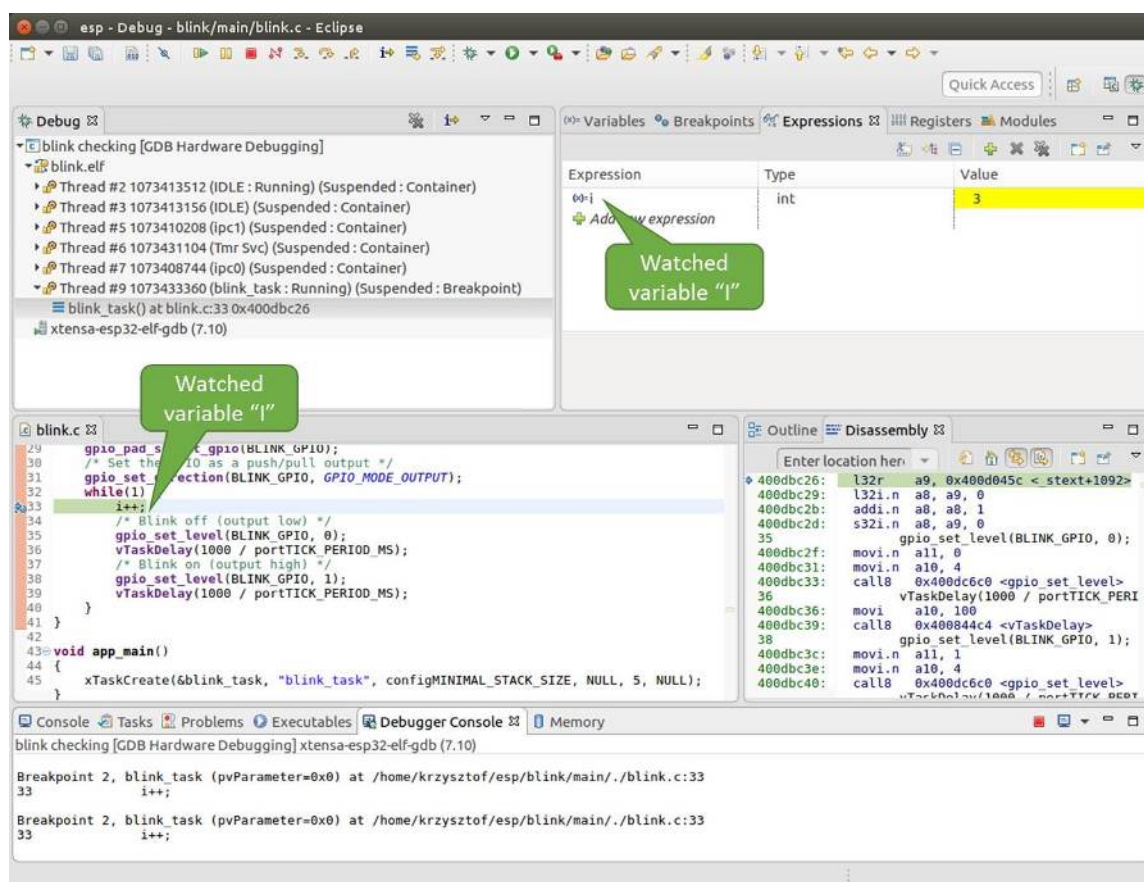


Fig. 52: Watching program variable "i"

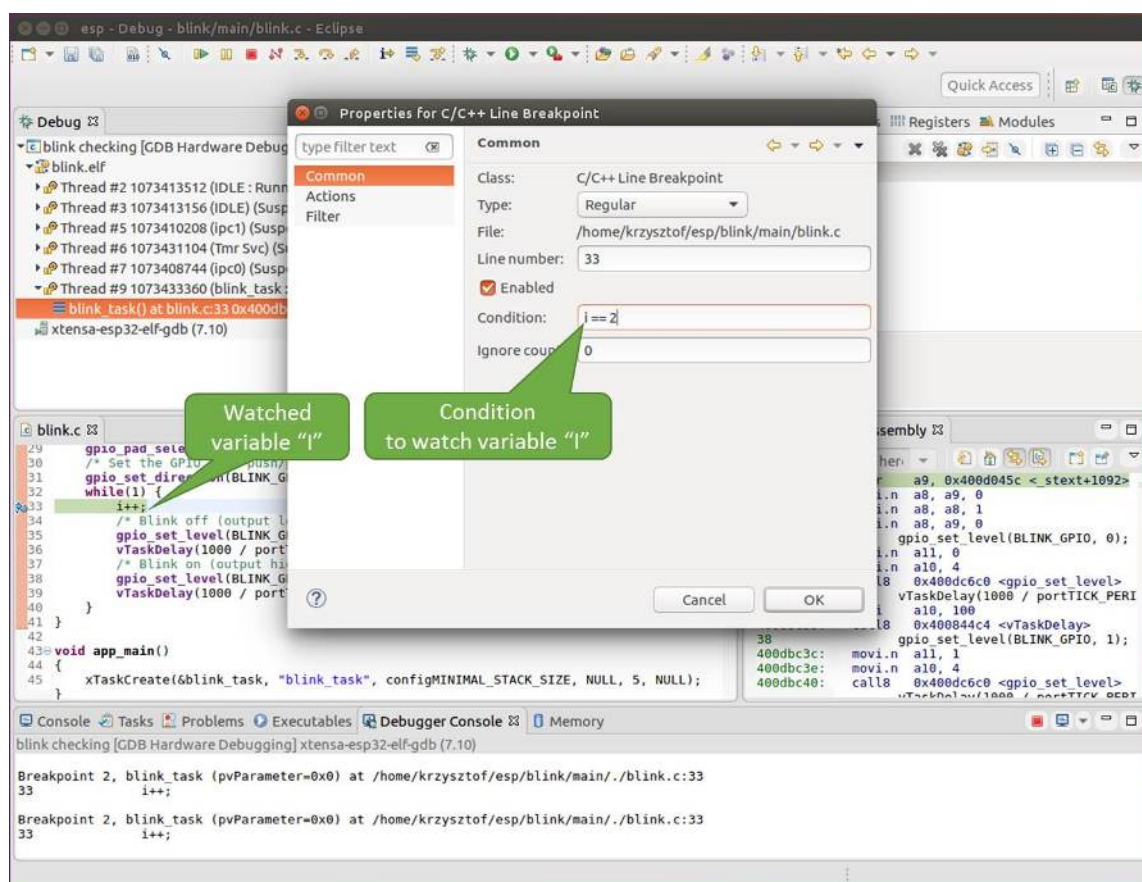


Fig. 53: Setting a conditional breakpoint

3. *Halting and resuming the application*
4. *Stepping through the code*
5. *Checking and setting memory*
6. *Watching and setting program variables*
7. *Setting conditional breakpoints*

Navigating through the code, call stack and threads When you see the (gdb) prompt, the application is halted. LED should not be blinking.

To find out where exactly the code is halted, enter `l` or `list`, and debugger will show couple of lines of code around the halt point (line 43 of code in file `blink.c`)

```
(gdb) l
38     }
39   }
40
41   void app_main()
42   {
43     xTaskCreate(&blink_task, "blink_task", configMINIMAL_STACK_SIZE, NULL, 5, &
↳NULL);
44   }
(gdb)
```

Check how code listing works by entering, e.g. `l 30, 40` to see particular range of lines of code.

You can use `bt` or `backtrace` to see what function calls lead up to this code:

```
(gdb) bt
#0  app_main () at /home/user-name/esp/blink/main/./blink.c:43
#1  0x400d057e in main_task (args=0x0) at /home/user-name/esp/esp-idf/components/
↳esp32/./cpu_start.c:339
(gdb)
```

Line #0 of output provides the last function call before the application halted, i.e. `app_main ()` we have listed previously. The `app_main ()` was in turn called by function `main_task` from line 339 of code located in file `cpu_start.c`.

To get to the context of `main_task` in file `cpu_start.c`, enter `frame N`, where `N = 1`, because the `main_task` is listed under #1):

```
(gdb) frame 1
#1  0x400d057e in main_task (args=0x0) at /home/user-name/esp/esp-idf/components/
↳esp32/./cpu_start.c:339
339   app_main();
(gdb)
```

Enter `l` and this will reveal the piece of code that called `app_main ()` (in line 339):

```
(gdb) l
334     ;
335   }
336 #endif
337   //Enable allocation in region where the startup stacks were located.
338   heap_caps_enable_nonos_stack_heaps();
339   app_main();
340   vTaskDelete(NULL);
341 }
342
(gdb)
```

By listing some lines before, you will see the function name `main_task` we have been looking for:

```
(gdb) l 326, 341
326 static void main_task(void* args)
327 {
328     // Now that the application is about to start, disable boot watchdogs
329     REG_CLR_BIT(TIMG_WDTCONFIG0_REG(0), TIMG_WDT_FLASHBOOT_MOD_EN_S);
330     REG_CLR_BIT(RTC_CNTL_WDTCONFIG0_REG, RTC_CNTL_WDT_FLASHBOOT_MOD_EN);
331 #if !CONFIG_FREERTOS_UNICORE
332     // Wait for FreeRTOS initialization to finish on APP CPU, before replacing
↳its startup stack
333     while (port_xSchedulerRunning[1] == 0) {
334         ;
335     }
336 #endif
337 //Enable allocation in region where the startup stacks were located.
338 heap_caps_enable_nonos_stack_heaps();
339 app_main();
340 vTaskDelete(NULL);
341 }
(gdb)
```

To see the other code, enter `i threads`. This will show the list of threads running on target:

```
(gdb) i threads
  Id  Target Id      Frame
  8   Thread 1073411336 (dport) 0x400d0848 in dport_access_init_core (arg=
↳<optimized out>)
    at /home/user-name/esp/esp-idf/components/esp32/./dport_access.c:170
  7   Thread 1073408744 (ipc0) xQueueGenericReceive (xQueue=0x3ffae694,
↳pvBuffer=0x0, xTicksToWait=1644638200,
    xJustPeeking=0) at /home/user-name/esp/esp-idf/components/freertos/./queue.
↳c:1452
  6   Thread 1073431096 (Tmr Svc) prvTimerTask (pvParameters=0x0)
    at /home/user-name/esp/esp-idf/components/freertos/./timers.c:445
  5   Thread 1073410208 (ipc1 : Running) 0x4000bfea in ?? ()
  4   Thread 1073432224 (dport) dport_access_init_core (arg=0x0)
    at /home/user-name/esp/esp-idf/components/esp32/./dport_access.c:150
  3   Thread 1073413156 (IDLE) prvIdleTask (pvParameters=0x0)
    at /home/user-name/esp/esp-idf/components/freertos/./tasks.c:3282
  2   Thread 1073413512 (IDLE) prvIdleTask (pvParameters=0x0)
    at /home/user-name/esp/esp-idf/components/freertos/./tasks.c:3282
* 1   Thread 1073411772 (main : Running) app_main () at /home/user-name/esp/blink/
↳main/./blink.c:43
(gdb)
```

The thread list shows the last function calls per each thread together with the name of C source file if available.

You can navigate to specific thread by entering `thread N`, where `N` is the thread Id. To see how it works go to thread thread 5:

```
(gdb) thread 5
[Switching to thread 5 (Thread 1073410208)]
#0 0x4000bfea in ?? ()
(gdb)
```

Then check the backtrace:

```
(gdb) bt
#0 0x4000bfea in ?? ()
#1 0x40083a85 in vPortCPUReleaseMutex (mux=<optimized out>) at /home/user-name/
↳esp/esp-idf/components/freertos/./port.c:415
#2 0x40083fc8 in vTaskSwitchContext () at /home/user-name/esp/esp-idf/components/
↳freertos/./tasks.c:2846
```

(continues on next page)

(continued from previous page)

```
#3 0x4008532b in _frxt_dispatch ()
#4 0x4008395c in xPortStartScheduler () at /home/user-name/esp/esp-idf/components/
↳freertos/./port.c:222
#5 0x4000000c in ?? ()
#6 0x4000000c in ?? ()
#7 0x4000000c in ?? ()
#8 0x4000000c in ?? ()
(gdb)
```

As you see, the backtrace may contain several entries. This will let you check what exact sequence of function calls lead to the code where the target halted. Question marks ?? instead of a function name indicate that application is available only in binary format, without any source file in C language. The value like 0x4000bfea is the memory address of the function call.

Using `bt`, `i threads`, `thread N` and `list` commands we are now able to navigate through the code of entire application. This comes handy when stepping through the code and working with breakpoints and will be discussed below.

Setting and clearing breakpoints When debugging, we would like to be able to stop the application at critical lines of code and then examine the state of specific variables, memory and registers / peripherals. To do so we are using breakpoints. They provide a convenient way to quickly get to and halt the application at specific line.

Let's establish two breakpoints when the state of LED changes. Basing on code listing above this happens at lines 33 and 36. Breakpoints may be established using command `break M` where `M` is the code line number:

```
(gdb) break 33
Breakpoint 2 at 0x400db6f6: file /home/user-name/esp/blink/main/./blink.c, line 33.
(gdb) break 36
Breakpoint 3 at 0x400db704: file /home/user-name/esp/blink/main/./blink.c, line 36.
```

If you now enter `c`, the processor will run and halt at a breakpoint. Entering `c` another time will make it run again, halt on second breakpoint, and so on:

```
(gdb) c
Continuing.
Target halted. PRO_CPU: PC=0x400DB6F6 (active) APP_CPU: PC=0x400D10D8

Breakpoint 2, blink_task (pvParameter=0x0) at /home/user-name/esp/blink/main/./
↳blink.c:33
33         gpio_set_level(BLINK_GPIO, 0);
(gdb) c
Continuing.
Target halted. PRO_CPU: PC=0x400DB6F8 (active) APP_CPU: PC=0x400D10D8
Target halted. PRO_CPU: PC=0x400DB704 (active) APP_CPU: PC=0x400D10D8

Breakpoint 3, blink_task (pvParameter=0x0) at /home/user-name/esp/blink/main/./
↳blink.c:36
36         gpio_set_level(BLINK_GPIO, 1);
(gdb)
```

You will be also able to see that LED is changing the state only if you resume program execution by entering `c`.

To examine how many breakpoints are set and where, use command `info break`:

```
(gdb) info break
Num      Type          Disp Enb Address      What
2        breakpoint     keep y   0x400db6f6 in blink_task at /home/user-name/esp/
↳blink/main/./blink.c:33
         breakpoint already hit 1 time
3        breakpoint     keep y   0x400db704 in blink_task at /home/user-name/esp/
↳blink/main/./blink.c:36
```

(continues on next page)

(continued from previous page)

```
breakpoint already hit 1 time
(gdb)
```

Please note that breakpoint numbers (listed under Num) start with 2. This is because first breakpoint has been already established at function `app_main()` by running command `thb app_main` on debugger launch. As it was a temporary breakpoint, it has been automatically deleted and now is not listed anymore.

To remove breakpoints enter `delete N` command (in short `d N`), where `N` is the breakpoint number:

```
(gdb) delete 1
No breakpoint number 1.
(gdb) delete 2
(gdb)
```

Read more about breakpoints under [Breakpoints and watchpoints available](#) and [What else should I know about breakpoints?](#)

Halting and resuming the application When debugging, you may resume application and enter code waiting for some event or staying in infinite loop without any break points defined. In such case, to go back to debugging mode, you can break program execution manually by entering `Ctrl+C`.

To check it delete all breakpoints and enter `c` to resume application. Then enter `Ctrl+C`. Application will be halted at some random point and LED will stop blinking. Debugger will print the following:

```
(gdb) c
Continuing.
^CTarget halted. PRO_CPU: PC=0x400D0C00          APP_CPU: PC=0x400D0C00 (active)
[New Thread 1073433352]

Program received signal SIGINT, Interrupt.
[Switching to Thread 1073413512]
0x400d0c00 in esp_vApplicationIdleHook () at /home/user-name/esp/esp-idf/
↳components/esp32/./freertos_hooks.c:52
52          asm("waiti 0");
(gdb)
```

In particular case above, the application has been halted in line 52 of code in file `freertos_hooks.c`. Now you can resume it again by enter `c` or do some debugging as discussed below.

Note: In MSYS2 shell `Ctrl+C` does not halt the target but exists debugger. To resolve this issue consider debugging with [Eclipse](#) or check a workaround under http://www.mingw.org/wiki/Workaround_for_GDB_Ctrl_C_Interrupt.

Stepping through the code It is also possible to step through the code using `step` and `next` commands (in short `s` and `n`). The difference is that `step` is entering inside subroutines calls, while `next` steps over the call, treating it as a single source line.

To demonstrate this functionality, using command `break` and `delete` discussed in previous paragraph, make sure that you have only one breakpoint defined at line 36 of `blink.c`:

```
(gdb) info break
Num      Type           Disp Enb Address      What
3        breakpoint     keep y   0x400db704  in blink_task at /home/user-name/esp/
↳blink/main/./blink.c:36
breakpoint already hit 1 time
(gdb)
```

Resume program by entering `c` and let it halt:

```
(gdb) c
Continuing.
Target halted. PRO_CPU: PC=0x400DB754 (active)    APP_CPU: PC=0x400D1128

Breakpoint 3, blink_task (pvParameter=0x0) at /home/user-name/esp/blink/main/. /
↳blink.c:36
36          gpio_set_level(BLINK_GPIO, 1);
(gdb)
```

Then enter `n` couple of times to see how debugger is stepping one program line at a time:

```
(gdb) n
Target halted. PRO_CPU: PC=0x400DB756 (active)    APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400DB758 (active)    APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400DC04C (active)    APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400DB75B (active)    APP_CPU: PC=0x400D1128
37          vTaskDelay(1000 / portTICK_PERIOD_MS);
(gdb) n
Target halted. PRO_CPU: PC=0x400DB75E (active)    APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400846FC (active)    APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400DB761 (active)    APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400DB746 (active)    APP_CPU: PC=0x400D1128
33          gpio_set_level(BLINK_GPIO, 0);
(gdb)
```

If you enter `s` instead, then debugger will step inside subroutine calls:

```
(gdb) s
Target halted. PRO_CPU: PC=0x400DB748 (active)    APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400DB74B (active)    APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400DC04C (active)    APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400DC04F (active)    APP_CPU: PC=0x400D1128
gpio_set_level (gpio_num=GPIO_NUM_4, level=0) at /home/user-name/esp/esp-idf/
↳components/driver/./gpio.c:183
183      GPIO_CHECK(GPIO_IS_VALID_OUTPUT_GPIO(gpio_num), "GPIO output gpio_num error
↳", ESP_ERR_INVALID_ARG);
(gdb)
```

In this particular case debugger stepped inside `gpio_set_level(BLINK_GPIO, 0)` and effectively moved to `gpio.c` driver code.

See *Why stepping with “next” does not bypass subroutine calls?* for potential limitation of using `next` command.

Checking and setting memory Displaying the contents of memory is done with command `x`. With additional parameters you may vary the format and count of memory locations displayed. Run `help x` to see more details. Companion command to `x` is `set` that let you write values to the memory.

We will demonstrate how `x` and `set` work by reading from and writing to the memory location `0x3FF44004` labeled as `GPIO_OUT_REG` used to set and clear individual GPIO's. For more information please refer to [ESP32 Technical Reference Manual](#), chapter `IO_MUX` and `GPIO Matrix`.

Being in the same `blink.c` project as before, set two breakpoints right after `gpio_set_level` instruction. Enter two times `c` to get to the break point followed by `x /1wx 0x3FF44004` to display contents of `GPIO_OUT_REG` memory location:

```
(gdb) c
Continuing.
Target halted. PRO_CPU: PC=0x400DB75E (active)    APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400DB74E (active)    APP_CPU: PC=0x400D1128

Breakpoint 2, blink_task (pvParameter=0x0) at /home/user-name/esp/blink/main/. /
↳blink.c:34
```

(continues on next page)

(continued from previous page)

```

34         vTaskDelay(1000 / portTICK_PERIOD_MS);
(gdb) x /1wx 0x3FF44004
0x3ff44004: 0x00000000
(gdb) c
Continuing.
Target halted. PRO_CPU: PC=0x400DB751 (active)     APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400DB75B (active)     APP_CPU: PC=0x400D1128

Breakpoint 3, blink_task (pvParameter=0x0) at /home/user-name/esp/blink/main/./
↳blink.c:37
37         vTaskDelay(1000 / portTICK_PERIOD_MS);
(gdb) x /1wx 0x3FF44004
0x3ff44004: 0x00000010
(gdb)

```

If you are blinking LED connected to GPIO4, then you should see fourth bit being flipped each time the LED changes the state:

```

0x3ff44004: 0x00000000
...
0x3ff44004: 0x00000010

```

Now, when the LED is off, that corresponds to `0x3ff44004: 0x00000000` being displayed, try using `set` command to set this bit by writing `0x00000010` to the same memory location:

```

(gdb) x /1wx 0x3FF44004
0x3ff44004: 0x00000000
(gdb) set {unsigned int}0x3FF44004=0x000010

```

You should see the LED to turn on immediately after entering `set {unsigned int}0x3FF44004=0x000010` command.

Watching and setting program variables A common debugging task is checking the value of a program variable as the program runs. To be able to demonstrate this functionality, update file `blink.c` by adding a declaration of a global variable `int i` above definition of function `blink_task`. Then add `i++` inside `while(1)` of this function to get `i` incremented on each blink.

Exit debugger, so it is not confused with new code, build and flash the code to the ESP and restart debugger. There is no need to restart OpenOCD.

Once application is halted, enter the command `watch i`:

```

(gdb) watch i
Hardware watchpoint 2: i
(gdb)

```

This will insert so called “watchpoint” in each place of code where variable `i` is being modified. Now enter `continue` to resume the application and observe it being halted:

```

(gdb) c
Continuing.
Target halted. PRO_CPU: PC=0x400DB751 (active)     APP_CPU: PC=0x400D0811
[New Thread 1073432196]

Program received signal SIGTRAP, Trace/breakpoint trap.
[Switching to Thread 1073432196]
0x400db751 in blink_task (pvParameter=0x0) at /home/user-name/esp/blink/main/./
↳blink.c:33
33         i++;
(gdb)

```


Resume application couple more times so `i` gets incremented. Now you can enter `print i` (in short `p i`) to check the current value of `i`:

```
(gdb) p i
$1 = 3
(gdb)
```

To modify the value of `i` use `set` command as below (you can then print it out to check if it has been indeed changed):

```
(gdb) set var i = 0
(gdb) p i
$3 = 0
(gdb)
```

You may have up to two watchpoints, see [Breakpoints and watchpoints available](#).

Setting conditional breakpoints Here comes more interesting part. You may set a breakpoint to halt the program execution, if certain condition is satisfied. Delete existing breakpoints and try this:

```
(gdb) break blink.c:34 if (i == 2)
Breakpoint 3 at 0x400db753: file /home/user-name/esp/blink/main/./blink.c, line 34.
(gdb)
```

Above command sets conditional breakpoint to halt program execution in line 34 of `blink.c` if `i == 2`.

If current value of `i` is less than 2 and program is resumed, it will blink LED in a loop until condition `i == 2` gets true and then finally halt:

```
(gdb) set var i = 0
(gdb) c
Continuing.
Target halted. PRO_CPU: PC=0x400DB755 (active) APP_CPU: PC=0x400D112C
Target halted. PRO_CPU: PC=0x400DB753 (active) APP_CPU: PC=0x400D112C
Target halted. PRO_CPU: PC=0x400DB755 (active) APP_CPU: PC=0x400D112C
Target halted. PRO_CPU: PC=0x400DB753 (active) APP_CPU: PC=0x400D112C

Breakpoint 3, blink_task (pvParameter=0x0) at /home/user-name/esp/blink/main/./
↪blink.c:34
34      gpio_set_level(BLINK_GPIO, 0);
(gdb)
```

Obtaining help on commands Commands presented so far should provide a very basic and intended to let you quickly get started with JTAG debugging. Check help what are the other commands at your disposal. To obtain help on syntax and functionality of a particular command, being at the `(gdb)` prompt type `help` and command name:

```
(gdb) help next
Step program, proceeding through subroutine calls.
Usage: next [N]
Unlike "step", if the current source line calls a subroutine,
this command does not enter the subroutine, but instead steps over
the call, in effect treating it as a single source line.
(gdb)
```

By typing just `help`, you will get a top-level list of command classes, to aid you in drilling down to more details. Optionally refer to available GDB cheat sheets, for instance <http://darkdust.net/files/GDB%20Cheat%20Sheet.pdf>. Good to have as a reference (even if not all commands are applicable in an embedded environment).

Ending debugger session To quit the debugger enter `q`:

```
(gdb) q
A debugging session is active.

    Inferior 1 [Remote target] will be detached.

Quit anyway? (y or n) y
Detaching from program: /home/user-name/esp/blink/build/blink.elf, Remote target
Ending remote debugging.
user-name@computer-name:~/esp/blink$
```

Tips and Quirks

This section provides collection of all tips and quirks referred to from various parts of this guide.

Breakpoints and watchpoints available ESP32 debugger supports 2 hardware implemented breakpoints and 64 software ones. Hardware breakpoints are implemented by ESP32 chip's logic and can be set anywhere in the code: either in flash or IRAM program's regions. Additionally there are 2 types of software breakpoints implemented by OpenOCD: flash (up to 32) and IRAM (up to 32) breakpoints. Currently GDB can not set software breakpoints in flash. So until this limitation is removed those breakpoints have to be emulated by OpenOCD as hardware ones (see [below](#) for details). ESP32 also supports two watchpoints, so two variables can be watched for change or read by the GDB command `watch myVariable`. Note that menuconfig option `CONFIG_FREERTOS_WATCHPOINT_END_OF_STACK` uses the 2nd watchpoint and will not provide expected results, if you also try to use it within OpenOCD / GDB. See menuconfig's help for detailed description.

What else should I know about breakpoints? Emulating part of hardware breakpoints using software flash ones means that the GDB command `hb myFunction` which is invoked for function in flash will use pure hardware breakpoint if it is available otherwise one of the 32 software flash breakpoints is used. The same rule applies to `b myFunction`-like commands. In this case GDB will decide what type of breakpoint to set itself. If `myFunction` is resided in writable region (IRAM) software IRAM breakpoint will be used otherwise hardware or software flash breakpoint is used as it is done for `hb` command.

Flash Mappings vs SW Flash Breakpoints In order to set/clear software breakpoints in flash, OpenOCD needs to know their flash addresses. To accomplish conversion from the ESP32 address space to the flash one, OpenOCD uses mappings of program's code regions resided in flash. Those mappings are kept in the image header which is prepended to program binary data (code and data segments) and is specific to every application image written to the flash. So to support software flash breakpoints OpenOCD should know where application image under debugging is resided in the flash. By default OpenOCD reads partition table at 0x8000 and uses mappings from the first found application image, but there can be the cases when it will not work, e.g. partition table is not at standard flash location or even there can be multiple images: one factory and two OTA and you may want to debug any of them. To cover all possible debugging scenarios OpenOCD supports special command which can be used to set arbitrary location of application image to debug. The command has the following format:

```
esp32 appimage_offset <offset>
```

Offset should be in hex format. To reset to the default behaviour you can specify `-1` as offset.

Note: Since GDB requests memory map from OpenOCD only once when connecting to it, this command should be specified in one of the TCL configuration files, or passed to OpenOCD via its command line. In the latter case command line should look like below:

```
openocd -f board/esp32-wrover-kit-3.3v.cfg -c "init; halt; esp32 appimage_offset 0x210000"
```

Another option is to execute that command via OpenOCD telnet session and then connect GDB, but it seems to be less handy.

Why stepping with “next” does not bypass subroutine calls? When stepping through the code with `next` command, GDB is internally setting a breakpoint (one out of two available) ahead in the code to bypass the subroutine calls. This functionality will not work, if the two available breakpoints are already set elsewhere in the code. If this is the case, delete breakpoints to have one “spare”. With both breakpoints already used, stepping through the code with `next` command will work as like with `step` command and debugger will step inside subroutine calls.

Support options for OpenOCD at compile time ESP-IDF has some support options for OpenOCD debugging which can be set at compile time:

- `CONFIG_ESP32_DEBUG_OCDAWARE` is enabled by default. If a panic or unhandled exception is thrown and a JTAG debugger is connected (ie OpenOCD is running), ESP-IDF will break into the debugger.
- `CONFIG_FREERTOS_WATCHPOINT_END_OF_STACK` (disabled by default) sets watchpoint index 1 (the second of two) at the end of any task stack. This is the most accurate way to debug task stack overflows. Click the link for more details.

Please see the [project configuration menu](#) menu for more details on setting compile-time options.

FreeRTOS support OpenOCD has explicit support for the ESP-IDF FreeRTOS. GDB can see FreeRTOS tasks as threads. Viewing them all can be done using the GDB `i threads` command, changing to a certain task is done with `thread n`, with `n` being the number of the thread. FreeRTOS detection can be disabled in target’s configuration. For more details see [Configuration of OpenOCD for specific target](#).

Why to set SPI flash voltage in OpenOCD configuration? The MTDI pin of ESP32, being among four pins used for JTAG communication, is also one of ESP32’s bootstrapping pins. On power up ESP32 is sampling binary level on MTDI to set it’s internal voltage regulator used to supply power to external SPI flash chip. If binary level on MTDI pin on power up is low, the voltage regulator is set to deliver 3.3 V, if it is high, then the voltage is set to 1.8 V. The MTDI pin should have a pull-up or may rely on internal weak pull down resistor (see [ESP32 Series Datasheet](#) for details), depending on the type of SPI chip used. Once JTAG is connected, it overrides the pull-up or pull-down resistor that is supposed to do the bootstrapping.

To handle this issue OpenOCD’s board configuration file (e.g. `boards\esp-wroom-32.cfg` for ESP32-WROOM-32 module) provides `ESP32_FLASH_VOLTAGE` parameter to set the idle state of the TDO line to a specified binary level, therefore reducing the chance of a bad bootup of application due to incorrect flash voltage.

Check specification of ESP32 module connected to JTAG, what is the power supply voltage of SPI flash chip. Then set `ESP32_FLASH_VOLTAGE` accordingly. Most WROOM modules use 3.3 V flash, while WROVER modules use 1.8 V flash.

Optimize JTAG speed In order to achieve higher data rates and minimize number of dropped packets it is recommended to optimize setting of JTAG clock frequency, so it is at maximum and still provides stable operation of JTAG. To do so use the following tips.

1. The upper limit of JTAG clock frequency is 20 MHz if CPU runs at 80 MHz, or 26 MHz if CPU runs at 160 MHz or 240 MHz.
2. Depending on particular JTAG adapter and the length of connecting cables, you may need to reduce JTAG frequency below 20 / 26 MHz.
3. In particular reduce frequency, if you get DSR/DIR errors (and they do not relate to OpenOCD trying to read from a memory range without physical memory being present there).
4. ESP-WROVER-KIT operates stable at 20 / 26 MHz.

What is the meaning of debugger’s startup commands? On startup, debugger is issuing sequence of commands to reset the chip and halt it at specific line of code. This sequence (shown below) is user defined to pick up at most convenient / appropriate line and start debugging.

- `set remote hardware-watchpoint-limit 2` —Restrict GDB to using two hardware watchpoints supported by ESP32. For more information see <https://sourceware.org/gdb/onlinedocs/gdb/Remote-Configuration.html>.
- `mon reset halt` —reset the chip and keep the CPUs halted

- `flushregs` —`monitor` (`mon`) command can not inform GDB that the target state has changed. GDB will assume that whatever stack the target had before `mon reset halt` will still be valid. In fact, after reset the target state will change, and executing `flushregs` is a way to force GDB to get new state from the target.
- `thb app_main` —insert a temporary hardware breakpoint at `app_main`, put here another function name if required
- `c` —resume the program. It will then stop at breakpoint inserted at `app_main`.

Configuration of OpenOCD for specific target OpenOCD needs to be told what JTAG adapter to use and processor the JTAG adapter is connected to. To do so, use existing **board** configuration files located in OpenOCD's `share/openocd/scripts/board` folder.

For example, if you connect to ESP-WROVER-KIT with ESP-WROOM-32 module installed (see section [ESP-WROVER-KIT V1 / ESP32 DevKitJ V1](#)), use the following configuration files:

- `board/esp32-wrover-kit-3.3v.cfg`

Optionally prepare configuration by yourself. To do so, you can check existing files and modify them to match your specific hardware. Below is the summary of available configuration parameters for **board** configuration.

Adapter's clock speed

```
adapter_khz 20000
```

See [Optimize JTAG speed](#) for guidance how to set this value.

Single core debugging

```
set ESP32_ONLYCPU 1
```

Comment out this line for dual core debugging.

Disable RTOS support

```
set ESP32_RTOS none
```

Comment out this line to have RTOS support.

Power supply voltage of ESP32's SPI flash chip

```
set ESP32_FLASH_VOLTAGE 1.8
```

Comment out this line to set 3.3 V, ref: [Why to set SPI flash voltage in OpenOCD configuration?](#)

Configuration file for ESP32 targets

```
source [find target/esp32.cfg]
```

Note: Do not change `source [find target/esp32.cfg]` line unless you are familiar with OpenOCD internals.

Currently `target/esp32.cfg` remains the only configuration file for ESP32 targets (`esp108` and `esp32`). The matrix of supported configurations is as follows:

| Dual/single | RTOS | Target used |
|-------------|----------|-------------|
| dual | FreeRTOS | esp32 |
| single | FreeRTOS | esp108 (*) |
| dual | none | esp108 |
| single | none | esp108 |

(*) —we plan to fix this and add support for single core debugging with esp32 target in a subsequent commits.

Look inside `board/esp-wroom-32.cfg` for additional information provided in comments besides each configuration parameter.

How debugger resets ESP32? The board can be reset by entering `mon reset` or `mon reset halt` into GDB.

Do not use JTAG pins for something else Operation of JTAG may be disturbed, if some other h/w is connected to JTAG pins besides ESP32 module and JTAG adapter. ESP32 JTAG is using the following pins:

| | ESP32 JTAG Pin | JTAG Signal |
|---|----------------|-------------|
| 1 | MTDO / GPIO15 | TDO |
| 2 | MTDI / GPIO12 | TDI |
| 3 | MTCK / GPIO13 | TCK |
| 4 | MTMS / GPIO14 | TMS |

JTAG communication will likely fail, if configuration of JTAG pins is changed by user application. If OpenOCD initializes correctly (detects the two Tensilica cores), but loses sync and spews out a lot of DTR/DIR errors when the program is ran, it is likely that the application reconfigures the JTAG pins to something else, or the user forgot to connect Vtar to a JTAG adapter that needed it.

Below is an excerpt from series of errors reported by GDB after the application stepped into the code that reconfigured MTDO / GPIO15 to be an input:

```
cpu0: xtensa_resume (line 431): DSR (FFFFFFFF) indicates target still busy!
cpu0: xtensa_resume (line 431): DSR (FFFFFFFF) indicates DIR instruction generated
↳an exception!
cpu0: xtensa_resume (line 431): DSR (FFFFFFFF) indicates DIR instruction generated
↳an overrun!
cpu1: xtensa_resume (line 431): DSR (FFFFFFFF) indicates target still busy!
cpu1: xtensa_resume (line 431): DSR (FFFFFFFF) indicates DIR instruction generated
↳an exception!
cpu1: xtensa_resume (line 431): DSR (FFFFFFFF) indicates DIR instruction generated
↳an overrun!
```

JTAG with Flash Encryption or Secure Boot By default, enabling Flash Encryption and/or Secure Boot will disable JTAG debugging. On first boot, the bootloader will burn an eFuse bit to permanently disable JTAG at the same time it enables the other features.

The project configuration option `CONFIG_SECURE_BOOT_ALLOW_JTAG` will keep JTAG enabled at this time, removing all physical security but allowing debugging. (Although the name suggests Secure Boot, this option can be applied even when only Flash Encryption is enabled).

However, OpenOCD may attempt to automatically read and write the flash in order to set *software breakpoints*. This has two problems:

- Software breakpoints are incompatible with Flash Encryption, OpenOCD currently has no support for encrypting or decrypting flash contents.
- If Secure Boot is enabled, setting a software breakpoint will change the digest of a signed app and make the signature invalid. This means if a software breakpoint is set and then a reset occurs, the signature verification will fail on boot.

To disable software breakpoints while using JTAG, add an extra argument `-c 'set ESP_FLASH_SIZE 0'` to the start of the OpenOCD command line. For example:

```
openocd -c 'set ESP_FLASH_SIZE 0' -f board/esp32-wrover-kit-3.3v.cfg
```

Note: For the same reason, the ESP-IDF app may fail bootloader verification of app signatures, when this option is enabled and a software breakpoint is set.

JTAG and ESP32-WROOM-32 AT firmware Compatibility Issue The ESP32-WROOM series of modules come pre-flashed with AT firmware. This firmware configures the pins GPIO12 to GPIO15 as SPI slave interface, which makes using JTAG impossible.

To make JTAG available, build new firmware that is not using pins GPIO12 to GPIO15 dedicated to JTAG communication. After that, flash the firmware onto your module. See also *Do not use JTAG pins for something else*.

Reporting issues with OpenOCD / GDB In case you encounter a problem with OpenOCD or GDB programs itself and do not find a solution searching available resources on the web, open an issue in the OpenOCD issue tracker under <https://github.com/espressif/openocd-esp32/issues>.

1. In issue report provide details of your configuration:
 - a. JTAG adapter type.
 - b. Release of ESP-IDF used to compile and load application that is being debugged.
 - c. Details of OS used for debugging.
 - d. Is OS running natively on a PC or on a virtual machine?
2. Create a simple example that is representative to observed issue. Describe steps how to reproduce it. In such an example debugging should not be affected by non-deterministic behaviour introduced by the Wi-Fi stack, so problems will likely be easier to reproduce, if encountered once.
3. Prepare logs from debugging session by adding additional parameters to start up commands.
OpenOCD:

```
openocd -l openocd_log.txt -d3 -f board/esp32-wrover-kit-3.3v.cfg
```

Logging to a file this way will prevent information displayed on the terminal. This may be a good thing taken amount of information provided, when increased debug level `-d3` is set. If you still like to see the log on the screen, then use another command instead:

```
openocd -d3 -f board/esp32-wrover-kit-3.3v.cfg 2>&1 | tee openocd.log
```

Debugger:

```
xtensa-esp32-elf-gdb -ex "set remotelogfile gdb_log.txt" <all other_
↪options>
```

Optionally add command `remotelogfile gdb_log.txt` to the `gdbinit` file.

4. Attach both `openocd_log.txt` and `gdb_log.txt` files to your issue report.
 - [Using Debugger](#)
 - [Debugging Examples](#)
 - [Tips and Quirks](#)
 - [Application Level Tracing library](#)
 - [Introduction to ESP-Prog Board](#)

5.20 Linker Script Generation

5.20.1 Overview

There are several *memory regions* where code and data can be placed. Code and read-only data are placed by default in flash, writable data in RAM, etc. However, it is sometimes necessary to change these default placements. For example, it may be necessary to place critical code in RAM for performance reasons or to place code in RTC memory for use in a wake stub or the ULP coprocessor.

With the linker script generation mechanism, it is possible to specify these placements at the component level within ESP-IDF. The component presents information on how it would like to place its symbols, objects or the entire archive. During build the information presented by the components are collected, parsed and processed; and the placement rules generated is used to link the app.

5.20.2 Quick Start

This section presents a guide for quickly placing code/data to RAM and RTC memory - placements ESP-IDF provides out-of-the-box.

For this guide, suppose we have the following:

```

- components/
  - my_component/
    - CMakeLists.txt
    - component.mk
    - Kconfig
    - src/
      - my_src1.c
      - my_src2.c
      - my_src3.c
    - my_linker_fragment_file.lf

```

- a component named `my_component` that is archived as library `libmy_component.a` during build
- three source files archived under the library, `my_src1.c`, `my_src2.c` and `my_src3.c` which are compiled as `my_src1.o`, `my_src2.o` and `my_src3.o`, respectively
- under `my_src1.o`, the function `my_function1` is defined; under `my_src2.o`, the function `my_function2` is defined
- there exist bool-type config `PERFORMANCE_MODE` (y/n) and int type config `PERFORMANCE_LEVEL` (with range 0-3) in `my_component`'s `Kconfig`

Creating and Specifying a Linker Fragment File

Before anything else, a linker fragment file needs to be created. A linker fragment file is simply a text file with a `.lf` extension upon which the desired placements will be written. After creating the file, it is then necessary to present it to the build system. The instructions for the build systems supported by ESP-IDF are as follows:

Make In the component's `component.mk` file, set the variable `COMPONENT_ADD_LDFRAGMENTS` to the path of the created linker fragment file. The path can either be an absolute path or a relative path from the component directory.

```
COMPONENT_ADD_LDFRAGMENTS += my_linker_fragment_file.lf
```

CMake In the component's `CMakeLists.txt` file, specify argument `LDFRAGMENTS` in the `idf_component_register` call. The value of `LDFRAGMENTS` can either be an absolute path or a relative path from the component directory to the created linker fragment file.

```

# file paths relative to CMakeLists.txt
idf_component_register(...
    LDFRAGMENTS "path/to/linker_fragment_file.lf" "path/to/
↪another_linker_fragment_file.lf"
    ...
)

```


Specifying placements

It is possible to specify placements at the following levels of granularity:

- object file (.obj or .o files)
- symbol (function/variable)
- archive (.a files)

Placing object files Suppose the entirety of `my_src1.o` is performance-critical, so it is desirable to place it in RAM. On the other hand, the entirety of `my_src2.o` contains symbols needed coming out of deep sleep, so it needs to be put under RTC memory. In the linker fragment file, we can write:

```
[mapping:my_component]
archive: libmy_component.a
entries:
    my_src1 (noflash)      # places all my_src1 code/read-only data under IRAM/DRAM
    my_src2 (rtc)         # places all my_src2 code/ data and read-only data under_
↳RTC fast memory/RTC slow memory
```

What happens to `my_src3.o`? Since it is not specified, default placements are used for `my_src3.o`. More on default placements [here](#).

Placing symbols Continuing our example, suppose that among functions defined under `object1.o`, only `my_function1` is performance-critical; and under `object2.o`, only `my_function2` needs to execute after the chip comes out of deep sleep. This could be accomplished by writing:

```
[mapping:my_component]
archive: libmy_component.a
entries:
    my_src1:my_function1 (noflash)
    my_src2:my_function2 (rtc)
```

The default placements are used for the rest of the functions in `my_src1.o` and `my_src2.o` and the entire `object3.o`. Something similar can be achieved for placing data by writing the variable name instead of the function name, like so:

```
my_src1:my_variable (noflash)
```

Warning: There are *limitations* in placing code/data at symbol granularity. In order to ensure proper placements, an alternative would be to group relevant code and data into source files, and *use object-granularity placements*.

Placing entire archive In this example, suppose that the entire component archive needs to be placed in RAM. This can be written as:

```
[mapping:my_component]
archive: libmy_component.a
entries:
    * (noflash)
```

Similarly, this places the entire component in RTC memory:

```
[mapping:my_component]
archive: libmy_component.a
entries:
    * (rtc)
```


Configuration-dependent placements Suppose that the entire component library should only have special placement when a certain condition is true; for example, when `CONFIG_PERFORMANCE_MODE == y`. This could be written as:

```
[mapping:my_component]
archive: libmy_component.a
entries:
    if PERFORMANCE_MODE = y:
        * (noflash)
    else:
        * (default)
```

For a more complex config-dependent placement, suppose the following requirements: when `CONFIG_PERFORMANCE_LEVEL == 1`, only `object1.o` is put in RAM; when `CONFIG_PERFORMANCE_LEVEL == 2`, `object1.o` and `object2.o`; and when `CONFIG_PERFORMANCE_LEVEL == 3` all object files under the archive are to be put into RAM. When these three are false however, put entire library in RTC memory. This scenario is a bit contrived, but, it can be written as:

```
[mapping:my_component]
archive: libmy_component.a
entries:
    if PERFORMANCE_LEVEL = 1:
        my_src1 (noflash)
    elif PERFORMANCE_LEVEL = 2:
        my_src1 (noflash)
        my_src2 (noflash)
    elif PERFORMANCE_LEVEL = 3:
        my_src1 (noflash)
        my_src2 (noflash)
        my_src3 (noflash)
    else:
        * (rtc)
```

Nesting condition-checking is also possible. The following is equivalent to the snippet above:

```
[mapping:my_component]
archive: libmy_component.a
entries:
    if PERFORMANCE_LEVEL <= 3 && PERFORMANCE_LEVEL > 0:
        if PERFORMANCE_LEVEL >= 1:
            object1 (noflash)
            if PERFORMANCE_LEVEL >= 2:
                object2 (noflash)
                if PERFORMANCE_LEVEL >= 3:
                    object2 (noflash)
    else:
        * (rtc)
```

The ‘default’ placements

Up until this point, the term ‘default placements’ has been mentioned as fallback placements for when the placement rules `rtc` and `noflash` are not specified. It is important to note that the tokens `noflash` or `rtc` are not merely keywords, but are actually entities called fragments, specifically *schemes*.

In the same manner as `rtc` and `noflash` are schemes, there exists a `default` scheme which defines what the default placement rules should be. As the name suggests, it is where code and data are usually placed, i.e. code/constants is placed in flash, variables placed in RAM, etc. More on the default scheme [here](#).

Note: For an example of an ESP-IDF component using the linker script generation mechanism, see [freer](#)-

[tos/CMakeLists.txt](#). `freertos` uses this to place its object files to the instruction RAM for performance reasons.

This marks the end of the quick start guide. The following text discusses the internals of the mechanism in a little bit more detail. The following sections should be helpful in creating custom placements or modifying default behavior.

5.20.3 Linker Script Generation Internals

Linking is the last step in the process of turning C/C++ source files into an executable. It is performed by the toolchain's linker, and accepts linker scripts which specify code/data placements, among other things. With the linker script generation mechanism, this process is no different, except that the linker script passed to the linker is dynamically generated from: (1) the collected *linker fragment files* and (2) *linker script template*.

Note: The tool that implements the linker script generation mechanism lives under [tools/ldgen](#).

Linker Fragment Files

As mentioned in the quick start guide, fragment files are simple text files with the `.lf` extension containing the desired placements. This is a simplified description of what fragment files contain, however. What fragment files actually contain are 'fragments'. Fragments are entities which contain pieces of information which, when put together, form placement rules that tell where to place sections of object files in the output binary. There are three types of fragments: *sections*, *scheme* and *mapping*.

Grammar The three fragment types share a common grammar:

```
[type:name]
key: value
key:
  value
  value
  value
  ...
```

- **type:** Corresponds to the fragment type, can either be `sections`, `scheme` or `mapping`.
- **name:** The name of the fragment, should be unique for the specified fragment type.
- **key, value:** Contents of the fragment; each fragment type may support different keys and different grammars for the key values.

Note: In cases where multiple fragments of the same type and name are encountered, an exception is thrown.

Note: The only valid characters for fragment names and keys are alphanumeric characters and underscore.

Condition Checking

Condition checking enable the linker script generation to be configuration-aware. Depending on whether expressions involving configuration values are true or not, a particular set of values for a key can be used. The evaluation uses `eval_string` from [tools/kconfig_new/kconfiglib.py](#) and adheres to its required syntax and limitations. Supported operators are as follows:

- **comparison**
 - `LessThan <`
 - `LessThanOrEqualTo <=`
 - `MoreThan >`
 - `MoreThanOrEqualTo >=`

- Equal =
- NotEqual !=
- **logical**
 - Or ||
 - And &&
 - Negation !
- **grouping**
 - Parenthesis ()

Condition checking behaves as you would expect an `if...elseif/elif...else` block in other languages. Condition-checking is possible for both key values and entire fragments. The two sample fragments below are equivalent:

```
# Value for keys is dependent on config
[type:name]
key_1:
    if CONDITION = y:
        value_1
    else:
        value_2
key_2:
    if CONDITION = y:
        value_a
    else:
        value_b
```

```
# Entire fragment definition is dependent on config
if CONDITION = y:
    [type:name]
    key_1:
        value_1
    key_2:
        value_b
else:
    [type:name]
    key_1:
        value_2
    key_2:
        value_b
```

Comments

Comment in linker fragment files begin with #. Like in other languages, comment are used to provide helpful descriptions and documentation and are ignored during processing.

Compatibility with ESP-IDF v3.x Linker Script Fragment Files ESP-IDF v4.0 brings some changes to the linker script fragment file grammar:

- **indentation is enforced and improperly indented fragment files generate a parse exception; this was not enforced in the** and examples demonstrates properly indented grammar
- move to `if...elif...else` structure for conditionals, with the ability to nest checks and place entire fragments themselves inside conditionals
- mapping fragments now requires a name like other fragment types

Linker script generator should be able to parse ESP-IDF v3.x linker fragment files that are indented properly (as demonstrated by the ESP-IDF v3.x version of this document). Backward compatibility with the previous mapping fragment grammar (optional name and the old grammar for conditionals) has also been retained but with a deprecation warning. Users should switch to the newer grammar discussed in this document as support for the old grammar is planned to be removed in the future.

Note that linker fragment files using the new ESP-IDF v4.0 grammar is not supported on ESP-IDF v3.x, however.

Types Sections

Sections fragments defines a list of object file sections that the GCC compiler emits. It may be a default section (e.g. `.text`, `.data`) or it may be user defined section through the `__attribute__` keyword.

The use of an optional `'+'` indicates the inclusion of the section in the list, as well as sections that start with it. This is the preferred method over listing both explicitly.

```
[sections:name]
entries:
    .section+
    .section
    ...
```

Example:

```
# Non-preferred
[sections:text]
entries:
    .text
    .text.*
    .literal
    .literal.*

# Preferred, equivalent to the one above
[sections:text]
entries:
    .text+           # means .text and .text.*
    .literal+       # means .literal and .literal.*
```

Scheme

Scheme fragments define what `target` a sections fragment is assigned to.

```
[scheme:name]
entries:
    sections -> target
    sections -> target
    ...
```

Example:

```
[scheme:noflash]
entries:
    text -> iram0_text           # the entries under the sections fragment named_
↔text will go to iram0_text
    rodata -> dram0_data       # the entries under the sections fragment named_
↔rodata will go to dram0_data
```

The default scheme

There exists a special scheme with the name `default`. This scheme is special because catch-all placement rules are generated from its entries. This means that, if one of its entries is `text -> flash_text`, the placement rule

```
*(.literal .literal.* .text .text.*)
```

will be generated for the target `flash_text`.

These catch-all rules then effectively serve as fallback rules for those whose mappings were not specified.

Note: The `default` scheme is defined in `esp32/ld/esp32_fragments.lf`. The `noflash` and `rtc` scheme fragments which are built-in schemes referenced in the quick start guide are also defined in this file.

Mapping

Mapping fragments define what scheme fragment to use for mappable entities, i.e. object files, function names, variable names, archives.

```
[mapping:name]
archive: archive          # output archive file name, as built (i.e. libxxx.
↳a)
entries:
  object:symbol (scheme)  # symbol granularity
  object (scheme)        # object granularity
  * (scheme)             # archive granularity
```

There are three levels of placement granularity:

- **symbol:** The object file name and symbol name are specified. The symbol name can be a function name or a variable name.
- **object:** Only the object file name is specified.
- **archive:** * is specified, which is a short-hand for all the object files under the archive.

To know what an entry means, let us expand a sample object-granularity placement:

```
object (scheme)
```

Then expanding the scheme fragment from its entries definitions, we have:

```
object (sections -> target,
        sections -> target,
        ...)
```

Expanding the sections fragment with its entries definition:

```
object (.section,      # given this object file
        .section,     # put its sections listed here at this
        ... -> target, # target

        .section,
        .section,     # same should be done for these sections
        ... -> target,

        ...)          # and so on
```

Example:

```
[mapping:map]
archive: libfreertos.a
entries:
  * (noflash)
```

On Symbol-Granularity Placements Symbol granularity placements is possible due to compiler flags – `ffunction-sections` and `ffdata-sections`. ESP-IDF compiles with these flags by default. If the user opts to remove these flags, then the symbol-granularity placements will not work. Furthermore, even with the presence of these flags, there are still other limitations to keep in mind due to the dependence on the compiler’s emitted output sections.

For example, with `ffunction-sections`, separate sections are emitted for each function; with section names predictably constructed i.e. `.text.{func_name}` and `.literal.{func_name}`. This is not the case for string literals within the function, as they go to pooled or generated section names.

With `fddata-sections`, for global scope data the compiler predictably emits either `.data.{var_name}`, `.rodata.{var_name}` or `.bss.{var_name}`; and so Type I mapping entry works for these. However, this is not the case for static data declared in function scope, as the generated section name is a result of mangling the variable name with some other information.

Linker Script Template

The linker script template is the skeleton in which the generated placement rules are put into. It is an otherwise ordinary linker script, with a specific marker syntax that indicates where the generated placement rules are placed.

To reference the placement rules collected under a `target` token, the following syntax is used:

```
mapping[target]
```

Example:

The example below is an excerpt from a possible linker script template. It defines an output section `.iram0.text`, and inside is a marker referencing the target `iram0_text`.

```
.iram0.text :
{
    /* Code marked as running out of IRAM */
    _iram_text_start = ABSOLUTE(.);

    /* Marker referencing iram0_text */
    mapping[iram0_text]

    _iram_text_end = ABSOLUTE(.);
} > iram0_0_seg
```

Suppose the generator collected the fragment definitions below:

```
[sections:text]
    .text+
    .literal+

[sections:iram]
    .iram1+

[scheme:default]
entries:
    text -> flash_text
    iram -> iram0_text

[scheme:noflash]
entries:
    text -> iram0_text

[mapping:freertos]
archive: libfreertos.a
entries:
    * (noflash)
```

Then the corresponding excerpt from the generated linker script will be as follows:

```
.iram0.text :
{
    /* Code marked as running out of IRAM */
    _iram_text_start = ABSOLUTE(.);

    /* Placement rules generated from the processed fragments, placed where the_
    ↪marker was in the template */
    *(.iram1 .iram1.*)
    *libfreertos.a:(.literal .text .literal.* .text.*)

    _iram_text_end = ABSOLUTE(.);
} > iram0_0_seg
```

```
*libfreertos.a:(.literal .text .literal.* .text.*)
```

Rule generated from the entry `* (noflash)` of the `freertos` mapping fragment. All `text` sections of all object files under the archive `libfreertos.a` will be collected under the target `iram0_text` (as per the `noflash` scheme) and placed wherever in the template `iram0_text` is referenced by a marker.

```
*(.iram1 .iram1.*)
```

Rule generated from the default scheme entry `iram -> iram0_text`. Since the default scheme specifies an `iram -> iram0_text` entry, it too is placed wherever `iram0_text` is referenced by a marker. Since it is a rule generated from the default scheme, it comes first among all other rules collected under the same target name.

Note: The linker script template currently used is <esp32/ld/esp32.project.ld.in>, specified by the `esp32` component; the generated output script is put under its build directory.

5.21 lwIP

ESP-IDF uses the open source [lwIP lightweight TCP/IP stack](#). The ESP-IDF version of lwIP ([esp-lwip](#)) has some modifications and additions compared to the upstream project.

5.21.1 Supported APIs

ESP-IDF supports the following lwIP TCP/IP stack functions:

- [BSD Sockets API](#)
- [Netconn API](#) is enabled but not officially supported for ESP-IDF applications

Adapted APIs

Some common lwIP “app” APIs are supported indirectly by ESP-IDF:

- DHCP Server & Client are supported indirectly via the [ESP-NETIF](#) functionality
- Simple Network Time Protocol (SNTP) is supported via the [lwip/include/apps/sntp/sntp.h](#) [lwip/lwip/src/include/lwip/apps/sntp.h](#) functions (see also [SNTP Time Synchronization](#))
- ICMP Ping is supported using a variation on the lwIP ping API. See [ICMP Echo](#).
- NetBIOS lookup is available using the standard lwIP API. [protocols/http_server/restful_server](#) has an option to demonstrate using NetBIOS to look up a host on the LAN.
- mDNS uses a different implementation to the lwIP default mDNS (see [mDNS Service](#)), but lwIP can look up mDNS hosts using standard APIs such as `gethostbyname()` and the convention `hostname.local`, provided the [CONFIG_LWIP_DNS_SUPPORT_MDNS_QUERIES](#) setting is enabled.

5.21.2 BSD Sockets API

The BSD Sockets API is a common cross-platform TCP/IP sockets API that originated in the Berkeley Standard Distribution of UNIX but is now standardized in a section of the POSIX specification. BSD Sockets are sometimes called POSIX Sockets or Berkeley Sockets.

As implemented in ESP-IDF, lwIP supports all of the common usages of the BSD Sockets API.

References

A wide range of BSD Sockets reference material is available, including:

- [Single UNIX Specification BSD Sockets page](#)
- [Berkeley Sockets Wikipedia page](#)

Examples

A number of ESP-IDF examples show how to use the BSD Sockets APIs:

- [protocols/sockets/tcp_server](#)
- [protocols/sockets/tcp_client](#)
- [protocols/sockets/udp_server](#)
- [protocols/sockets/udp_client](#)
- [protocols/sockets/udp_multicast](#)
- [protocols/http_request](#) (Note: this is a simplified example of using a TCP socket to send an HTTP request. The *ESP HTTP Client* is a much better option for sending HTTP requests.)

Supported functions

The following BSD socket API functions are supported. For full details see [lwip/lwip/src/include/lwip/sockets.h](#).

- `socket()`
- `bind()`
- `accept()`
- `shutdown()`
- `getpeername()`
- `getsockopt()` & `setsockopt()` (see [Socket Options](#))
- `close()` (via [Virtual filesystem component](#))
- `read()`, `readv()`, `write()`, `writew()` (via [Virtual filesystem component](#))
- `recv()`, `recvmsg()`, `recvfrom()`
- `send()`, `sendmsg()`, `sendto()`
- `select()` (via [Virtual filesystem component](#))
- `poll()` (Note: on ESP-IDF, `poll()` is implemented by calling `select` internally, so using `select()` directly is recommended if a choice of methods is available.)
- `fcntl()` (see [fcntl](#))

Non-standard functions:

- `ioctl()` (see [ioctls](#))

Note: Some lwIP application sample code uses prefixed versions of BSD APIs, for example `lwip_socket()` instead of the standard `socket()`. Both forms can be used with ESP-IDF, but using standard names is recommended.

Socket Error Handling

BSD Socket error handling code is very important for robust socket applications. Normally the socket error handling involves the following aspects:

- Detecting the error.
- Getting the error reason code.
- Handle the error according to the reason code.

In lwIP, we have two different scenarios of handling socket errors:

- Socket API returns an error. For more information, see [Socket API Errors](#).
- `select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset, struct timeval *timeout)` has exception descriptor indicating that the socket has an error. For more information, see [select\(\) Errors](#).

Socket API Errors

The error detection

- We can know that the socket API fails according to its return value.

Get the error reason code

- When socket API fails, the return value doesn't contain the failure reason and the application can get the error reason code by accessing `errno`. Different values indicate different meanings. For more information, see [<Socket Error Reason Code>](#).

Example:

```
int err;
int sockfd;

if (sockfd = socket(AF_INET, SOCK_STREAM, 0) < 0) {
    // the error code is obtained from errno
    err = errno;
    return err;
}
```

select() Errors

The error detection

- Socket error when `select()` has exception descriptor

Get the error reason code

- If the `select` indicates that the socket fails, we can't get the error reason code by accessing `errno`, instead we should call `getsockopt()` to get the failure reason code. Because `select()` has exception descriptor, the error code will not be given to `errno`.

Note: `getsockopt` function prototype `int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen)`. Its function is to get the current value of the option of any type, any state socket, and store the result in `optval`. For example, when you get the error code on a socket, you can get it by `getsockopt(sockfd, SOL_SOCKET, SO_ERROR, &err, &optlen)`.

Example:

```
int err;

if (select(sockfd + 1, NULL, NULL, &exfds, &tval) <= 0) {
    err = errno;
    return err;
} else {
    if (FD_ISSET(sockfd, &exfds)) {
        // select() exception set using getsockopt()
        int optlen = sizeof(int);
        getsockopt(sockfd, SOL_SOCKET, SO_ERROR, &err, &optlen);
        return err;
    }
}
```

Socket Error Reason Code Below is a list of common error codes. For more detailed error codes, please query [newlib/include/sys/errno.h](#)

| Error code | Description |
|-----------------|-------------------------------------|
| ECONNREFUSED | Connection refused |
| EADDRINUSE | Address already in use |
| ECONNABORTED | Software caused connection abort |
| ENETUNREACH | Network is unreachable |
| ENETDOWN | Network interface is not configured |
| ETIMEDOUT | Connection timed out |
| EHOSTDOWN | Host is down |
| EHOSTUNREACH | Host is unreachable |
| EINPROGRESS | Connection already in progress |
| EALREADY | Socket already connected |
| EDESTADDRREQ | Destination address required |
| EPROTONOSUPPORT | Unknown protocol |

Socket Options

The `getsockopt()` and `setsockopt()` functions allow getting/setting per-socket options.

Not all standard socket options are supported by lwIP in ESP-IDF. The following socket options are supported:

Common options Used with level argument `SOL_SOCKET`.

- `SO_REUSEADDR` (available if `CONFIG_LWIP_SO_REUSE` is set, behavior can be customized by setting `CONFIG_LWIP_SO_REUSE_RXTOALL`)
- `SO_KEEPALIVE`
- `SO_BROADCAST`
- `SO_ACCEPTCONN`
- `SO_RCVBUF` (available if `CONFIG_LWIP_SO_RCVBUF` is set)
- `SO_SNDTIMEO` / `SO_RCVTIMEO`
- `SO_ERROR` (this option is only used with `select()`, see [Socket Error Handling](#))
- `SO_TYPE`
- `SO_NO_CHECK` (for UDP sockets only)

IP options Used with level argument `IPPROTO_IP`.

- `IP_TOS`
- `IP_TTL`
- `IP_PKTINFO` (available if `CONFIG_LWIP_NETBUF_RECVINFO` is set)

For multicast UDP sockets:

- `IP_MULTICAST_IF`
- `IP_MULTICAST_LOOP`
- `IP_MULTICAST_TTL`
- `IP_ADD_MEMBERSHIP`
- `IP_DROP_MEMBERSHIP`

TCP options TCP sockets only. Used with level argument `IPPROTO_TCP`.

- `TCP_NODELAY`

Options relating to TCP keepalive probes:

- `TCP_KEEPALIVE` (int value, TCP keepalive period in milliseconds)
- `TCP_KEEPIDLE` (same as `TCP_KEEPALIVE`, but the value is in seconds)
- `TCP_KEEPINTVL` (int value, interval between keepalive probes in seconds)
- `TCP_KEEPCNT` (int value, number of keepalive probes before timing out)

IPv6 options IPv6 sockets only. Used with level argument `IPPROTO_IPV6`

- `IPV6_CHECKSUM`
- `IPV6_V6ONLY`

For multicast IPv6 UDP sockets:

- `IPV6_JOIN_GROUP / IPV6_ADD_MEMBERSHIP`
- `IPV6_LEAVE_GROUP / IPV6_DROP_MEMBERSHIP`
- `IPV6_MULTICAST_IF`
- `IPV6_MULTICAST_HOPS`
- `IPV6_MULTICAST_LOOP`

fcntl

The `fcntl()` function is a standard API for manipulating options related to a file descriptor. In ESP-IDF, the *Virtual filesystem component* layer is used to implement this function.

When the file descriptor is a socket, only the following `fcntl()` values are supported:

- `O_NONBLOCK` to set/clear non-blocking I/O mode. Also supports `O_NDELAY`, which is identical to `O_NONBLOCK`.
- `O_RDONLY`, `O_WRONLY`, `O_RDWR` flags for different read/write modes. These can read via `F_GETFL` only, they cannot be set using `F_SETFL`. A TCP socket will return a different mode depending on whether the connection has been closed at either end or is still open at both ends. UDP sockets always return `O_RDWR`.

ioctl

The `ioctl()` function provides a semi-standard way to access some internal features of the TCP/IP stack. In ESP-IDF, the *Virtual filesystem component* layer is used to implement this function.

When the file descriptor is a socket, only the following `ioctl()` values are supported:

- `FIONREAD` returns the number of bytes of pending data already received in the socket's network buffer.
- `FIONBIO` is an alternative way to set/clear non-blocking I/O status for a socket, equivalent to `fcntl(fd, F_SETFL, O_NONBLOCK, ...)`.

5.21.3 Netconn API

lwIP supports two lower level APIs as well as the BSD Sockets API: the Netconn API and the Raw API.

The lwIP Raw API is designed for single threaded devices and is not supported in ESP-IDF.

The Netconn API is used to implement the BSD Sockets API inside lwIP, and it can also be called directly from ESP-IDF apps. This API has lower resource usage than the BSD Sockets API, in particular it can send and receive data without needing to first copy it into internal lwIP buffers.

Important: Espressif does not test the Netconn API in ESP-IDF. As such, this functionality is *enabled but not supported*. Some functionality may only work correctly when used from the BSD Sockets API.

For more information about the Netconn API, consult [lwip/lwip/src/include/lwip/api.h](#) and [this wiki page which is part of the unofficial lwIP Application Developers Manual](#).

5.21.4 lwIP FreeRTOS Task

lwIP creates a dedicated TCP/IP FreeRTOS task to handle socket API requests from other tasks.

A number of configuration items are available to modify the task and the queues (“mailboxes”) used to send data to/from the TCP/IP task:

- [*CONFIG_LWIP_TCPIP_RECVMBOX_SIZE*](#)
- [*CONFIG_LWIP_TCPIP_TASK_STACK_SIZE*](#)
- [*CONFIG_LWIP_TCPIP_TASK_AFFINITY*](#)

5.21.5 esp-lwip custom modifications

Additions

The following code is added which is not present in the upstream lwIP release:

Thread-safe sockets It is possible to `close()` a socket from a different thread to the one that created it. The `close()` call will block until any function calls currently using that socket from other tasks have returned.

It is, however, not possible to delete a task while it is actively waiting on `select()` or `poll()` APIs. It is always necessary that these APIs exit before destroying the task, as this might corrupt internal structures and cause subsequent crashes of the lwIP. (These APIs allocate globally referenced callback pointers on stack, so that when the task gets destroyed before unrolling the stack, the lwIP would still hold pointers to the deleted stack)

On demand timers lwIP IGMP and MLD6 features both initialize a timer in order to trigger timeout events at certain times.

The default lwIP implementation is to have these timers enabled all the time, even if no timeout events are active. This increases CPU usage and power consumption when using automatic light sleep mode. `esp-lwip` default behaviour is to set each timer “on demand” so it is only enabled when an event is pending.

To return to the default lwIP behaviour (always-on timers), disable [*CONFIG_LWIP_TIMERS_ONDEMAND*](#).

Abort TCP connections when IP changes [*CONFIG_LWIP_TCP_KEEP_CONNECTION_WHEN_IP_CHANGES*](#) is disabled by default. This disables the default lwIP behaviour of keeping TCP connections open if an interface IP changes, in case the interface IP changes back (for example, if an interface connection goes down and comes back up). Enable this option to keep TCP connections open in this case, until they time out normally. This may increase the number of sockets in use if a network interface goes down temporarily.

Additional Socket Options

- Some standard IPV4 and IPV6 multicast socket options are implemented (see [*Socket Options*](#)).
- Possible to set IPV6-only UDP and TCP sockets with `IPV6_V6ONLY` socket option (normal lwIP is TCP only).

IP layer features

- IPV4 source based routing implementation is different.
- IPV4 mapped IPV6 addresses are supported.

Limitations

- Calling `send()` or `sendto()` repeatedly on a UDP socket may eventually fail with `errno` equal to `ENOMEM`. This is a limitation of buffer sizes in the lower layer network interface drivers. If all driver transmit buffers are full then UDP transmission will fail. Applications sending a high volume of UDP datagrams who don't wish for any to be dropped by the sender should check for this error code and re-send the datagram after a short delay. Increasing the number of TX buffers in the [*Wi-Fi*](#) or [*Ethernet*](#) project configuration (as applicable) may also help.

5.21.6 Performance Optimization

TCP/IP performance is a complex subject, and performance can be optimized towards multiple goals. The default settings of ESP-IDF are tuned for a compromise between throughput, latency, and moderate memory usage.

Maximum throughput

Espressif tests ESP-IDF TCP/IP throughput using the [wifi/iperf](#) example in an RF sealed enclosure.

The [wifi/iperf/sdkconfig.defaults](#) file for the iperf example contains settings known to maximize TCP/IP throughput, usually at the expense of higher RAM usage. To get maximum TCP/IP throughput in an application at the expense of other factors then suggest applying settings from this file into the project sdkconfig.

Important: Suggest applying changes a few at a time and checking the performance each time with a particular application workload.

- If a lot of tasks are competing for CPU time on the system, consider that the lwIP task has configurable CPU affinity ([CONFIG_LWIP_TCPIP_TASK_AFFINITY](#)) and runs at fixed priority `ESP_TASK_TCPIP_PRIO` (18). Configure competing tasks to be pinned to a different core, or to run at a lower priority.
- If using `select ()` function with socket arguments only, setting [CONFIG_LWIP_USE_ONLY_LWIP_SELECT](#) will make `select ()` calls faster.

If using a Wi-Fi network interface, please also refer to [Wi-Fi Buffer Usage](#).

Minimum latency

Except for increasing buffer sizes, most changes which increase throughput will also decrease latency by reducing the amount of CPU time spent in lwIP functions.

- For TCP sockets, lwIP supports setting the standard `TCP_NODELAY` flag to disable Nagle's algorithm.

Minimum RAM usage

Most lwIP RAM usage is on-demand, as RAM is allocated from the heap as needed. Therefore, changing lwIP settings to reduce RAM usage may not change RAM usage at idle but can change it at peak.

- Reducing [CONFIG_LWIP_MAX_SOCKETS](#) reduces the maximum number of sockets in the system. This will also cause TCP sockets in the `WAIT_CLOSE` state to be closed and recycled more rapidly (if needed to open a new socket), further reducing peak RAM usage.
- Reducing [CONFIG_LWIP_TCPIP_RECVMBOX_SIZE](#), [CONFIG_LWIP_TCP_RECVMBOX_SIZE](#) and [CONFIG_LWIP_UDP_RECVMBOX_SIZE](#) reduce memory usage at the expense of throughput, depending on usage.

If using Wi-Fi, please also refer to [Wi-Fi Buffer Usage](#).

Peak Buffer Usage The peak heap memory that lwIP consumes is the **theoretically-maximum memory** that the lwIP driver consumes. Generally, the peak heap memory that lwIP consumes depends on:

- the memory required to create a UDP connection: `lwip_udp_conn`
- the memory required to create a TCP connection: `lwip_tcp_conn`
- the number of UDP connections that the application has: `lwip_udp_con_num`
- the number of TCP connections that the application has: `lwip_tcp_con_num`
- the TCP TX window size: `lwip_tcp_tx_win_size`
- the TCP RX window size: `lwip_tcp_rx_win_size`

So, the peak heap memory that the LwIP consumes can be calculated with the following formula:

$$\text{lwip_dynamic_peek_memory} = (\text{lwip_udp_con_num} * \text{lwip_udp_conn}) + (\text{lwip_tcp_con_num} * (\text{lwip_tcp_tx_win_size} + \text{lwip_tcp_rx_win_size} + \text{lwip_tcp_conn}))$$

Some TCP-based applications need only one TCP connection. However, they may choose to close this TCP connection and create a new one when an error (such as a sending failure) occurs. This may result in multiple TCP connections existing in the system simultaneously, because it may take a long time for a TCP connection to close, according to the TCP state machine (refer to RFC793).

5.22 Partition Tables

5.22.1 Overview

A single ESP32's flash can contain multiple apps, as well as many different kinds of data (calibration data, filesystems, parameter storage, etc). For this reason a partition table is flashed to (*default offset*) 0x8000 in the flash.

Partition table length is 0xC00 bytes (maximum 95 partition table entries). An MD5 checksum, which is used for checking the integrity of the partition table, is appended after the table data. If the partition table is signed due to *secure boot*, the signature is appended after the partition table.

Each entry in the partition table has a name (label), type (app, data, or something else), subtype and the offset in flash where the partition is loaded.

The simplest way to use the partition table is to open the project configuration menu (`idf.py menuconfig`) and choose one of the simple predefined partition tables under `CONFIG_PARTITION_TABLE_TYPE`:

- “Single factory app, no OTA”
- “Factory app, two OTA definitions”

In both cases the factory app is flashed at offset 0x10000. If you execute `idf.py partition_table` then it will print a summary of the partition table.

5.22.2 Built-in Partition Tables

Here is the summary printed for the “Single factory app, no OTA” configuration:

```
# Espressif ESP32 Partition Table
# Name, Type, SubType, Offset, Size, Flags
nvs, data, nvs, 0x9000, 0x6000,
phy_init, data, phy, 0xf000, 0x1000,
factory, app, factory, 0x10000, 1M,
```

- At a 0x10000 (64KB) offset in the flash is the app labelled “factory” . The bootloader will run this app by default.
- There are also two data regions defined in the partition table for storing NVS library partition and PHY init data.

Here is the summary printed for the “Factory app, two OTA definitions” configuration:

```
# Espressif ESP32 Partition Table
# Name, Type, SubType, Offset, Size, Flags
nvs, data, nvs, 0x9000, 0x4000,
otadata, data, ota, 0xd000, 0x2000,
phy_init, data, phy, 0xf000, 0x1000,
factory, 0, 0, 0x10000, 1M,
ota_0, 0, ota_0, 0x110000, 1M,
ota_1, 0, ota_1, 0x210000, 1M,
```

- There are now three app partition definitions. The type of the factory app (at 0x10000) and the next two “OTA” apps are all set to “app” , but their subtypes are different.
- There is also a new “otadata” slot, which holds the data for OTA updates. The bootloader consults this data in order to know which app to execute. If “ota data” is empty, it will execute the factory app.

5.22.3 Creating Custom Tables

If you choose “Custom partition table CSV” in menuconfig then you can also enter the name of a CSV file (in the project directory) to use for your partition table. The CSV file can describe any number of definitions for the table you need.

The CSV format is the same format as printed in the summaries shown above. However, not all fields are required in the CSV. For example, here is the “input” CSV for the OTA partition table:

| # | Name, | Type, | SubType, | Offset, | Size, | Flags |
|---|-----------|-------|-----------|----------|--------|-------|
| | nvs, | data, | nvs, | 0x9000, | 0x4000 | |
| | otadata, | data, | ota, | 0xd000, | 0x2000 | |
| | phy_init, | data, | phy, | 0xf000, | 0x1000 | |
| | factory, | app, | factory, | 0x10000, | 1M | |
| | ota_0, | app, | ota_0, | , | 1M | |
| | ota_1, | app, | ota_1, | , | 1M | |
| | nvs_key, | data, | nvs_keys, | , | 0x1000 | |

- Whitespace between fields is ignored, and so is any line starting with # (comments).
- Each non-comment line in the CSV file is a partition definition.
- The “Offset” field for each partition is empty. The `gen_esp32part.py` tool fills in each blank offset, starting after the partition table and making sure each partition is aligned correctly.

Name field

Name field can be any meaningful name. It is not significant to the ESP32. Names longer than 16 characters will be truncated.

Type field

Partition type field can be specified as app (0) or data (1). Or it can be a number 0-254 (or as hex 0x00-0xFE). Types 0x00-0x3F are reserved for esp-idf core functions.

If your application needs to store data, please add a custom partition type in the range 0x40-0xFE.

The bootloader ignores any partition types other than app (0) & data (1).

SubType

The 8-bit subtype field is specific to a given partition type. esp-idf currently only specifies the meaning of the subtype field for “app” and “data” partition types.

- When type is “app”, the subtype field can be specified as factory (0), ota_0 (0x10) … ota_15 (0x1F) or test (0x20).
 - factory (0) is the default app partition. The bootloader will execute the factory app unless there it sees a partition of type data/ota, in which case it reads this partition to determine which OTA image to boot.
 - * OTA never updates the factory partition.
 - * If you want to conserve flash usage in an OTA project, you can remove the factory partition and use ota_0 instead.
 - ota_0 (0x10) … ota_15 (0x1F) are the OTA app slots. Refer to the [OTA documentation](#) for more details, which then use the OTA data partition to configure which app slot the bootloader should boot. If using OTA, an application should have at least two OTA application slots (ota_0 & ota_1). Refer to the [OTA documentation](#) for more details.
 - test (0x20) is a reserved subtype for factory test procedures. It will be used as the fallback boot partition if no other valid app partition is found. It is also possible to configure the bootloader to read a GPIO input during each boot, and boot this partition if the GPIO is held low, see [Boot from TEST firmware](#).
- When type is “data”, the subtype field can be specified as ota (0), phy (1), nvs (2), or nvs_keys (4).
 - ota (0) is the [OTA data partition](#) which stores information about the currently selected OTA application. This partition should be 0x2000 bytes in size. Refer to the [OTA documentation](#) for more details.

- phy (1) is for storing PHY initialisation data. This allows PHY to be configured per-device, instead of in firmware.
 - * In the default configuration, the phy partition is not used and PHY initialisation data is compiled into the app itself. As such, this partition can be removed from the partition table to save space.
 - * To load PHY data from this partition, open the project configuration menu (`idf.py menuconfig`) and enable `CONFIG_ESP32_PHY_INIT_DATA_IN_PARTITION` option. You will also need to flash your devices with phy init data as the esp-idf build system does not do this automatically.
- nvs (2) is for the *Non-Volatile Storage (NVS) API*.
 - * NVS is used to store per-device PHY calibration data (different to initialisation data).
 - * NVS is used to store WiFi data if the `esp_wifi_set_storage(WIFI_STORAGE_FLASH)` initialisation function is used.
 - * The NVS API can also be used for other application data.
 - * It is strongly recommended that you include an NVS partition of at least 0x3000 bytes in your project.
 - * If using NVS API to store a lot of data, increase the NVS partition size from the default 0x6000 bytes.
- nvs_keys (4) is for the NVS key partition. See *Non-Volatile Storage (NVS) API* for more details.
 - * It is used to store NVS encryption keys when *NVS Encryption* feature is enabled.
 - * The size of this partition should be 4096 bytes (minimum partition size).

Other data subtypes are reserved for future esp-idf uses.

Offset & Size

Partitions with blank offsets will start after the previous partition, or after the partition table in the case of the first partition.

App partitions have to be at offsets aligned to 0x10000 (64K). If you leave the offset field blank, `gen_esp32part.py` will automatically align the partition. If you specify an unaligned offset for an app partition, the tool will return an error.

Sizes and offsets can be specified as decimal numbers, hex numbers with the prefix 0x, or size multipliers K or M (1024 and 1024*1024 bytes).

If you want the partitions in the partition table to work with any starting offset (`CONFIG_PARTITION_TABLE_OFFSET`) of the table itself, leave the offset field (in CSV file) for all partitions blank. Similarly, if changing the partition table offset then be aware that all blank partition offsets may change to match, and that any fixed offsets may now collide with the partition table (causing an error).

Flags

Only one flag is currently supported, `encrypted`. If this field is set to `encrypted`, this partition will be encrypted if *Flash Encryption* is enabled.

(Note that app type partitions will always be encrypted, regardless of whether this flag is set or not.)

5.22.4 Generating Binary Partition Table

The partition table which is flashed to the ESP32 is in a binary format, not CSV. The tool `partition_table/gen_esp32part.py` is used to convert between CSV and binary formats.

If you configure the partition table CSV name in the project configuration (`idf.py menuconfig`) and then build the project or run `idf.py partition_table`, this conversion is done as part of the build process.

To convert CSV to Binary manually:

```
python gen_esp32part.py input_partitions.csv binary_partitions.bin
```

To convert binary format back to CSV manually:


```
python gen_esp32part.py binary_partitions.bin input_partitions.csv
```

To display the contents of a binary partition table on stdout (this is how the summaries displayed when running `idf.py partition_table` are generated:

```
python gen_esp32part.py binary_partitions.bin
```

MD5 checksum

The binary format of the partition table contains an MD5 checksum computed based on the partition table. This checksum is used for checking the integrity of the partition table during the boot.

The MD5 checksum generation can be disabled by the `--disable-md5sum` option of `gen_esp32part.py` or by the `CONFIG_PARTITION_TABLE_MD5` option. This is useful for example when one *uses a bootloader from ESP-IDF before v3.1* which cannot process MD5 checksums and the boot fails with the error message `invalid magic number 0xebeb`.

5.22.5 Flashing the partition table

- `idf.py partition_table-flash`: will flash the partition table with `esptool.py`.
- `idf.py flash`: Will flash everything including the partition table.

A manual flashing command is also printed as part of `idf.py partition_table` output.

Note that updating the partition table doesn't erase data that may have been stored according to the old partition table. You can use `idf.py erase_flash` (or `esptool.py erase_flash`) to erase the entire flash contents.

5.22.6 Partition Tool (`parttool.py`)

The component `partition_table` provides a tool `parttool.py` for performing partition-related operations on a target device. The following operations can be performed using the tool:

- reading a partition and saving the contents to a file (`read_partition`)
- writing the contents of a file to a partition (`write_partition`)
- erasing a partition (`erase_partition`)
- retrieving info such as offset and size of a given partition (`get_partition_info`)

The tool can either be imported and used from another Python script or invoked from shell script for users wanting to perform operation programmatically. This is facilitated by the tool's Python API and command-line interface, respectively.

Python API

Before anything else, make sure that the `parttool` module is imported.

```
import sys
import os

idf_path = os.environ["IDF_PATH"] # get value of IDF_PATH from environment
parttool_dir = os.path.join(idf_path, "components", "partition_table") # parttool.
↳ py lives in $IDF_PATH/components/partition_table

sys.path.append(parttool_dir) # this enables Python to find parttool module
from parttool import * # import all names inside parttool module
```

The starting point for using the tool's Python API to do is create a `ParttoolTarget` object:

```
# Create a partool.py target device connected on serial port /dev/ttyUSB1
target = ParttoolTarget("/dev/ttyUSB1")
```

The created object can now be used to perform operations on the target device:

```
# Erase partition with name 'storage'
target.erase_partition(PartitionName("storage"))

# Read partition with type 'data' and subtype 'spiffs' and save to file 'spiffs.bin'
↪
target.read_partition(PartitionType("data", "spiffs"), "spiffs.bin")

# Write to partition 'factory' the contents of a file named 'factory.bin'
target.write_partition(PartitionName("factory"), "factory.bin")

# Print the size of default boot partition
storage = target.get_partition_info(PARTITION_BOOT_DEFAULT)
print(storage.size)
```

The partition to operate on is specified using *PartitionName* or *PartitionType* or `PARTITION_BOOT_DEFAULT`. As the name implies, these can be used to refer to partitions of a particular name, type-subtype combination, or the default boot partition.

More information on the Python API is available in the docstrings for the tool.

Command-line Interface

The command-line interface of *partool.py* has the following structure:

```
partool.py [command-args] [subcommand] [subcommand-args]

- command-args - These are arguments that are needed for executing the main_
↪command (partool.py), mostly pertaining to the target device
- subcommand - This is the operation to be performed
- subcommand-args - These are arguments that are specific to the chosen operation
```

```
# Erase partition with name 'storage'
partool.py --port "/dev/ttyUSB1" erase_partition --partition-name=storage

# Read partition with type 'data' and subtype 'spiffs' and save to file 'spiffs.bin'
↪
partool.py --port "/dev/ttyUSB1" read_partition --partition-type=data --partition-
↪subtype=spiffs "spiffs.bin"

# Write to partition 'factory' the contents of a file named 'factory.bin'
partool.py --port "/dev/ttyUSB1" write_partition --partition-name=factory
↪"factory.bin"

# Print the size of default boot partition
partool.py --port "/dev/ttyUSB1" get_partition_info --partition-boot-default --
↪info size
```

More information can be obtained by specifying *-help* as argument:

```
# Display possible subcommands and show main command argument descriptions
partool.py --help

# Show descriptions for specific subcommand arguments
partool.py [subcommand] --help
```

5.23 RF calibration

ESP32 supports three RF calibration methods during RF initialization:

1. Partial calibration
2. Full calibration
3. No calibration

5.23.1 Partial calibration

During RF initialization, the partial calibration method is used by default for RF calibration. It is done based on the full calibration data which is stored in the NVS. To use this method, please go to `menuconfig` and enable [CONFIG_ESP32_PHY_CALIBRATION_AND_DATA_STORAGE](#).

5.23.2 Full calibration

Full calibration is triggered in the following conditions:

1. NVS does not exist.
2. The NVS partition to store calibration data is erased.
3. Hardware MAC address is changed.
4. PHY library version is changed.
5. The RF calibration data loaded from the NVS partition is broken.

It takes about 100ms more than partial calibration. If boot duration is not critical, it is suggested to use the full calibration method. To switch to the full calibration method, go to `menuconfig` and disable [CONFIG_ESP32_PHY_CALIBRATION_AND_DATA_STORAGE](#). If you use the default method of RF calibration, there are two ways to add the function of triggering full calibration as a last-resort remedy.

1. Erase the NVS partition if you don't mind all of the data stored in the NVS partition is erased. That is indeed the easiest way.
2. Call API `esp_phy_erase_cal_data_in_nvs()` before initializing WiFi and BT/BLE based on some conditions (e.g. an option provided in some diagnostic mode). In this case, only phy namespace of the NVS partition is erased.

5.23.3 No calibration

No calibration method is only used when ESP32 wakes up from deep sleep.

5.23.4 PHY initialization data

The PHY initialization data is used for RF calibration. There are two ways to get the PHY initialization data.

One is the default initialization data which is located in the header file [components/esp_wifi/esp32/include/phy_init_data.h](#). It is embedded into the application binary after compiling and then stored into read-only memory (DROM). To use the default initialization data, please go to `menuconfig` and disable [CONFIG_ESP32_PHY_INIT_DATA_IN_PARTITION](#).

Another is the initialization data which is stored in a partition. When using a custom partition table, make sure that PHY data partition is included (type: *data*, subtype: *phy*). With default partition table, this is done automatically. If initialization data is stored in a partition, it has to be flashed there, otherwise runtime error will occur. To switch to the initialization data stored in a partition, go to `menuconfig` and enable [CONFIG_ESP32_PHY_INIT_DATA_IN_PARTITION](#).

5.24 ESP32 ROM console

When an ESP32 is unable to boot from flash ROM (and the fuse disabling it hasn't been blown), it boots into a rom console. The console is based on TinyBasic, and statements entered should be in the form of BASIC statements. As is common in the BASIC language, without a preceding line number, commands entered are executed immediately; lines with a prefixed line number are stored as part of a program.

5.24.1 Full list of supported statements and functions

System

- **BYE** - *exits Basic, reboots ESP32, retries booting from flash*
- **END** - *stops execution from the program, also "STOP"*
- **MEM** - *displays memory usage statistics*
- **NEW** - *clears the current program*
- **RUN** - *executes the current program*

IO, Documentation

- **PEEK**(address) - *get a 32-bit value from a memory address*
- **POKE** - *write a 32-bit value to memory*
- **USR**(addr, arg1, ..) - *Execute a machine language function*
- **PRINT** expression - *print out the expression, also "?"*
- **PHEX** expression - *print expression as a hex number*
- **REM** stuff - *remark/comment, also ""*

Expressions, Math

- **A=V, LET A=V** - *assign value to a variable*
- **+, -, *, /** - *Math*
- **<, <=, =, <>, !=, >=, >** - *Comparisons*
- **ABS**(expression) - *returns the absolute value of the expression*
- **RSEED**(v) - *sets the random seed to v*
- **RND**(m) - *returns a random number from 0 to m*
- **A=1234** - ** Assign a decimal value**
- **A=&h1A2** - ** Assign a hex value**
- **A=&b1001** - *Assign a binary value*

Control

- **IF** expression statement - *perform statement if expression is true*
- **FOR** variable = start TO end - *start for block*
- **FOR** variable = start TO end STEP value - *start for block with step*
- **NEXT** - *end of for block*
- **GOTO** linenumber - *continue execution at this line number*
- **GOSUB** linenumber - *call a subroutine at this line number*
- **RETURN** - *return from a subroutine*
- **DELAY** - *Delay a given number of milliseconds*

Pin IO

- **IODIR** - *Set a GPIO-pin as an output (1) or input (0)*
- **IOSET** - *Set a GPIO-pin, configured as output, to high (1) or low (0)*
- **IOGET** - *Get the value of a GPIO-pin*

5.24.2 Example programs

Here are a few example commands and programs to get you started...

Read UART_DATE register of uart0

```
> PHEX PEEK(&h3FF40078)
15122500
```

Set GPIO2 using memory writes to GPIO_OUT_REG

Note: you can do this easier with the IOSET command

```
> POKE &h3FF44004, PEEK(&h3FF44004) OR &b100
```

Get value of GPIO0

```
> IODIR 0,0
> PRINT IOGET(0)
0
```

Blink LED

Hook up an LED between GPIO2 and ground. When running the program, the LED should blink 10 times.

```
10 IODIR 2,1
20 FOR A=1 TO 10
30 IOSET 2,1
40 DELAY 250
50 IOSET 2,0
60 DELAY 250
70 NEXT A
RUN
```

5.24.3 Credits

The ROM console is based on “TinyBasicPlus” by Mike Field and Scott Lawrence, which is based on “68000 TinyBasic” by Gordon Brandly

5.25 Secure Boot

Important: All references in this document are related to Secure Boot V1 (The AES based Secure Boot Scheme). ESP32 Revision 3 onwards, the preferred secure boot scheme is *Secure Boot V2*. Please refer to Secure Boot V2 document for ESP32 Revision 3.

Secure Boot is a feature for ensuring only your code can run on the chip. Data loaded from flash is verified on each reset.

Secure Boot is separate from the *Flash Encryption* feature, and you can use secure boot without encrypting the flash contents. However, for a secure environment both should be used simultaneously. See *Secure Boot & Flash Encryption* for more details.

Important: Enabling secure boot limits your options for further updates of your ESP32. Make sure to read this document thoroughly and understand the implications of enabling secure boot.

5.25.1 Background

- Most data is stored in flash. Flash access does not need to be protected from physical access in order for secure boot to function, because critical data is stored (non-software-accessible) in Efuses internal to the chip.
- Efuses are used to store the secure bootloader key (in efuse BLOCK2), and also a single Efuse bit (ABS_DONE_0) is burned (written to 1) to permanently enable secure boot on the chip. For more details about efuse, see Chapter 11 “eFuse Controller” in the Technical Reference Manual.
- To understand the secure boot process, first familiarise yourself with the standard *ESP-IDF boot process*.
- Both stages of the boot process (initial software bootloader load, and subsequent partition & app loading) are verified by the secure boot process, in a “chain of trust” relationship.

5.25.2 Secure Boot Process Overview

This is a high level overview of the secure boot process. Step by step instructions are supplied under *How To Enable Secure Boot*. Further in-depth details are supplied under *Technical Details*:

1. The options to enable secure boot are provided in the *Project Configuration Menu*, under “Secure Boot Configuration” .
2. Secure Boot defaults to signing images and partition table data during the build process. The “Secure boot private signing key” config item is a file path to a ECDSA public/private key pair in a PEM format file.
3. The software bootloader image is built by esp-idf with secure boot support enabled and the public key (signature verification) portion of the secure boot signing key compiled in. This software bootloader image is flashed at offset 0x1000.
4. On first boot, the software bootloader follows the following process to enable secure boot:
 - Hardware secure boot support generates a device secure bootloader key (generated via hardware RNG, then stored read/write protected in efuse), and a secure digest. The digest is derived from the key, an IV, and the bootloader image contents.
 - The secure digest is flashed at offset 0x0 in the flash.
 - Depending on Secure Boot Configuration, efuses are burned to disable JTAG and the ROM BASIC interpreter (it is strongly recommended these options are turned on.)
 - Bootloader permanently enables secure boot by burning the ABS_DONE_0 efuse. The software bootloader then becomes protected (the chip will only boot a bootloader image if the digest matches.)
5. On subsequent boots the ROM bootloader sees that the secure boot efuse is burned, reads the saved digest at 0x0 and uses hardware secure boot support to compare it with a newly calculated digest. If the digest does not match then booting will not continue. The digest and comparison are performed entirely by hardware, and the calculated digest is not readable by software. For technical details see *Secure Boot Hardware Support*.
6. When running in secure boot mode, the software bootloader uses the secure boot signing key (the public key of which is embedded in the bootloader itself, and therefore validated as part of the bootloader) to verify the signature appended to all subsequent partition tables and app images before they are booted.

5.25.3 Keys

The following keys are used by the secure boot process:

- “secure bootloader key” is a 256-bit AES key that is stored in Efuse block 2. The bootloader can generate this key itself from the internal hardware random number generator, the user does not need to supply it (it is optionally possible to supply this key, see *Re-Flashable Software Bootloader*). The Efuse holding this key is read & write protected (preventing software access) before secure boot is enabled.
 - By default, the Efuse Block 2 Coding Scheme is “None” and a 256 bit key is stored in this block. On some ESP32s, the Coding Scheme is set to 3/4 Encoding (CODING_SCHEME efuse has value 1) and a 192 bit key must be stored in this block. See ESP32 Technical Reference Manual section 20.3.1.3 *System*

Parameter coding_scheme for more details. The algorithm operates on a 256 bit key in all cases, 192 bit keys are extended by repeating some bits (*details*).

- “secure boot signing key” is a standard ECDSA public/private key pair (see *Image Signing Algorithm*) in PEM format.
 - The public key from this key pair (for signature verification but not signature creation) is compiled into the software bootloader and used to verify the second stage of booting (partition table, app image) before booting continues. The public key can be freely distributed, it does not need to be kept secret.
 - The private key from this key pair *must be securely kept private*, as anyone who has this key can authenticate to any bootloader that is configured with secure boot and the matching public key.

5.25.4 Bootloader Size

When secure boot is enabled the bootloader app binary `bootloader.bin` may exceed the default bootloader size limit. This is especially likely if flash encryption is enabled as well. The default size limit is 0x7000 (28672) bytes (partition table offset 0x8000 - bootloader offset 0x1000).

If the bootloader becomes too large, the ESP32 will fail to boot - errors will be logged about either invalid partition table or invalid bootloader checksum.

Options to work around this are:

- Reduce *bootloader log level*. Setting log level to Warning, Error or None all significantly reduce the final binary size (but may make it harder to debug).
- Set *partition table offset* to a higher value than 0x8000, to place the partition table later in the flash. This increases the space available for the bootloader. If the *partition table* CSV file contains explicit partition offsets, they will need changing so no partition has an offset lower than `CONFIG_PARTITION_TABLE_OFFSET + 0x1000`. (This includes the default partition CSV files supplied with ESP-IDF.)

5.25.5 How To Enable Secure Boot

1. Open the *Project Configuration Menu*, navigate to “Secure Boot Configuration” and select the option “One-time Flash” . (To understand the alternative “Reflashable” choice, see *Re-Flashable Software Bootloader*.)
2. Select a name for the secure boot signing key. This option will appear after secure boot is enabled. The file can be anywhere on your system. A relative path will be evaluated from the project directory. The file does not need to exist yet.
3. Set other menuconfig options (as desired). Pay particular attention to the “Bootloader Config” options, as you can only flash the bootloader once. Then exit menuconfig and save your configuration
4. The first time you run `make`, if the signing key is not found then an error message will be printed with a command to generate a signing key via `espsecure.py generate_signing_key`.

Important: A signing key generated this way will use the best random number source available to the OS and its Python installation (`/dev/urandom` on OSX/Linux and `CryptGenRandom()` on Windows). If this random number source is weak, then the private key will be weak.

Important: For production environments, we recommend generating the keypair using openssl or another industry standard encryption program. See *Generating Secure Boot Signing Key* for more details.

5. Run `idf.py bootloader` to build a secure boot enabled bootloader. The build output will include a prompt for a flashing command, using `esptool.py write_flash`.
6. When you’re ready to flash the bootloader, run the specified command (you have to enter it yourself, this step is not performed by `make`) and then wait for flashing to complete. **Remember this is a one time flash, you can’t change the bootloader after this!**
7. Run `idf.py flash` to build and flash the partition table and the just-built app image. The app image will be signed using the signing key you generated in step 4.

Note: `idf.py flash` doesn't flash the bootloader if secure boot is enabled.

8. Reset the ESP32 and it will boot the software bootloader you flashed. The software bootloader will enable secure boot on the chip, and then it verifies the app image signature and boots the app. You should watch the serial console output from the ESP32 to verify that secure boot is enabled and no errors have occurred due to the build configuration.

Note: Secure boot won't be enabled until after a valid partition table and app image have been flashed. This is to prevent accidents before the system is fully configured.

Note: If the ESP32 is reset or powered down during the first boot, it will start the process again on the next boot.

9. On subsequent boots, the secure boot hardware will verify the software bootloader has not changed (using the secure bootloader key) and then the software bootloader will verify the signed partition table and app image (using the public key portion of the secure boot signing key).

5.25.6 Re-Flashable Software Bootloader

Configuration “Secure Boot: One-Time Flash” is the recommended configuration for production devices. In this mode, each device gets a unique key that is never stored outside the device.

However, an alternative mode *Secure Boot: Reflashable* is also available. This mode allows you to supply a binary key file that is used for the secure bootloader key. As you have the key file, you can generate new bootloader images and secure boot digests for them.

In the `esp-idf` build process, this 256-bit key file is derived from the ECDSA app signing key generated by the user (see the *Generating Secure Boot Signing Key* step below). This private key's SHA-256 digest is used as the secure bootloader key in efuse (as-is for Coding Scheme None, or truncate to 192 bytes for 3/4 Encoding). This is a convenience so you only need to generate/protect a single private key.

Note: Although it's possible, we strongly recommend not generating one secure boot key and flashing it to every device in a production environment. The “One-Time Flash” option is recommended for production environments.

To enable a reflashable bootloader:

1. In the *Project Configuration Menu*, select “Bootloader Config” -> `CONFIG_SECURE_BOOT` -> `CONFIG_SECURE_BOOT_V1_ENABLED` -> `CONFIG_SECURE_BOOTLOADER_MODE` -> Reflashable.
2. If necessary, set the `CONFIG_SECURE_BOOTLOADER_KEY_ENCODING` based on the coding scheme used by the device. The coding scheme is shown in the `Features` line when `esptool.py` connects to the chip, or in the `espefuse.py summary` output.
2. Follow the steps shown above to choose a signing key file, and generate the key file.
3. Run `idf.py bootloader`. A binary key file will be created, derived from the private key that is used for signing. Two sets of flashing steps will be printed - the first set of steps includes an `espefuse.py burn_key` command which is used to write the bootloader key to efuse. (Flashing this key is a one-time-only process.) The second set of steps can be used to reflash the bootloader with a pre-calculated digest (generated during the build process).
4. Resume from *Step 6 of the one-time flashing process*, to flash the bootloader and enable secure boot. Watch the console log output closely to ensure there were no errors in the secure boot configuration.

5.25.7 Generating Secure Boot Signing Key

The build system will prompt you with a command to generate a new signing key via `espsecure.py generate_signing_key`. This uses the `python-ecdsa` library, which in turn uses Python's `os.urandom()` as a random number source.

The strength of the signing key is proportional to (a) the random number source of the system, and (b) the correctness of the algorithm used. For production devices, we recommend generating signing keys from a system with a quality entropy source, and using the best available EC key generation utilities.

For example, to generate a signing key using the `openssl` command line:

```
openssl ecparam -name prime256v1 -genkey -noout -out
my_secure_boot_signing_key.pem`
```

Remember that the strength of the secure boot system depends on keeping the signing key private.

5.25.8 Remote Signing of Images

For production builds, it can be good practice to use a remote signing server rather than have the signing key on the build machine (which is the default `esp-idf` secure boot configuration). The `espsecure.py` command line program can be used to sign app images & partition table data for secure boot, on a remote system.

To use remote signing, disable the option “Sign binaries during build”. The private signing key does not need to be present on the build system. However, the public (signature verification) key is required because it is compiled into the bootloader (and can be used to verify image signatures during OTA updates).

To extract the public key from the private key:

```
espsecure.py extract_public_key --keyfile PRIVATE_SIGNING_KEY PUBLIC_VERIFICATION_
↳KEY
```

The path to the public signature verification key needs to be specified in the `menuconfig` under “Secure boot public signature verification key” in order to build the secure bootloader.

After the app image and partition table are built, the build system will print signing steps using `espsecure.py`:

```
espsecure.py sign_data --keyfile PRIVATE_SIGNING_KEY BINARY_FILE
```

The above command appends the image signature to the existing binary. You can use the `-output` argument to write the signed binary to a separate file:

```
espsecure.py sign_data --keyfile PRIVATE_SIGNING_KEY --output SIGNED_BINARY_FILE_
↳BINARY_FILE
```

5.25.9 Secure Boot Best Practices

- Generate the signing key on a system with a quality source of entropy.
- Keep the signing key private at all times. A leak of this key will compromise the secure boot system.
- Do not allow any third party to observe any aspects of the key generation or signing process using `espsecure.py`. Both processes are vulnerable to timing or other side-channel attacks.
- Enable all secure boot options in the Secure Boot Configuration. These include flash encryption, disabling of JTAG, disabling BASIC ROM interpreter, and disabling the UART bootloader encrypted flash access.
- Use secure boot in combination with *flash encryption* to prevent local readout of the flash contents.

5.25.10 Technical Details

The following sections contain low-level reference descriptions of various secure boot elements:

Secure Boot Hardware Support

The first stage of secure boot verification (checking the software bootloader) is done via hardware. The ESP32's Secure Boot support hardware can perform three basic operations:

1. Generate a random sequence of bytes from a hardware random number generator.
2. Generate a digest from data (usually the bootloader image from flash) using a key stored in Efuse block 2. The key in Efuse can (& should) be read/write protected, which prevents software access. For full details of this algorithm see [Secure Bootloader Digest Algorithm](#). The digest can only be read back by software if Efuse ABS_DONE_0 is *not* burned (ie still 0).
3. Generate a digest from data (usually the bootloader image from flash) using the same algorithm as step 2 and compare it to a pre-calculated digest supplied in a buffer (usually read from flash offset 0x0). The hardware returns a true/false comparison without making the digest available to software. This function is available even when Efuse ABS_DONE_0 is burned.

Secure Bootloader Digest Algorithm

Starting with an “image” of binary data as input, this algorithm generates a digest as output. The digest is sometimes referred to as an “abstract” in hardware documentation.

For a Python version of this algorithm, see the `espsecure.py` tool in the `components/esptool_py` directory (specifically, the `digest_secure_bootloader` command).

Items marked with (^) are to fulfill hardware restrictions, as opposed to cryptographic restrictions.

1. Read the AES key from efuse block 2, in reversed byte order. If Coding Scheme is set to 3/4 Encoding, extend the 192 bit key to 256 bits using the same algorithm described in [Flash Encryption Algorithm](#).
2. Prefix the image with a 128 byte randomly generated IV.
3. If the image length is not modulo 128, pad the image to a 128 byte boundary with 0xFF. (^)
4. For each 16 byte plaintext block of the input image: - Reverse the byte order of the plaintext input block (^)
- Apply AES256 in ECB mode to the plaintext block. - Reverse the byte order of the ciphertext output block. (^) - Append to the overall ciphertext output.
5. Byte-swap each 4 byte word of the ciphertext (^)
6. Calculate SHA-512 of the ciphertext.

Output digest is 192 bytes of data: The 128 byte IV, followed by the 64 byte SHA-512 digest.

Image Signing Algorithm

Deterministic ECDSA as specified by [RFC 6979](#).

- Curve is NIST256p (openssl calls this curve “prime256v1”, it is also sometimes called secp256r1).
- Hash function is SHA256.
- Key format used for storage is PEM.
 - In the bootloader, the public key (for signature verification) is flashed as 64 raw bytes.
- Image signature is 68 bytes - a 4 byte version word (currently zero), followed by a 64 bytes of signature data. These 68 bytes are appended to an app image or partition table data.

Manual Commands

Secure boot is integrated into the esp-idf build system, so `make` will automatically sign an app image if secure boot is enabled. `idf.py bootloader` will produce a bootloader digest if `menuconfig` is configured for it.

However, it is possible to use the `espsecure.py` tool to make standalone signatures and digests.

To sign a binary image:

```
espsecure.py sign_data --keyfile ./my_signing_key.pem --output ./image_signed.bin  
↪image-unsigned.bin
```

Keyfile is the PEM file containing an ECDSA private signing key.

To generate a bootloader digest:

```
espsecure.py digest_secure_bootloader --keyfile ./securebootkey.bin --output ./  
↪bootloader-digest.bin build/bootloader/bootloader.bin
```

Keyfile is the 32 byte raw secure boot key for the device.

The output of the `espsecure.py digest_secure_bootloader` command is a single file which contains both the digest and the bootloader appended to it. To flash the combined digest plus bootloader to the device:

```
esptool.py write_flash 0x0 bootloader-digest.bin
```

5.25.11 Secure Boot & Flash Encryption

If secure boot is used without *Flash Encryption*, it is possible to launch “time-of-check to time-of-use” attack, where flash contents are swapped after the image is verified and running. Therefore, it is recommended to use both the features together.

5.25.12 Signed App Verification Without Hardware Secure Boot

The integrity of apps can be checked even without enabling the hardware secure boot option. This option uses the same app signature scheme as hardware secure boot, but unlike hardware secure boot it does not prevent the bootloader from being physically updated. This means that the device can be secured against remote network access, but not physical access. Compared to using hardware Secure Boot this option is much simpler to implement. See [How To Enable Signed App Verification](#) for step by step instructions.

An app can be verified on update and, optionally, be verified on boot.

- Verification on update: When enabled, the signature is automatically checked whenever the `esp_ota_ops.h` APIs are used for OTA updates. If hardware secure boot is enabled, this option is always enabled and cannot be disabled. If hardware secure boot is not enabled, this option still adds significant security against network-based attackers by preventing spoofing of OTA updates.
- Verification on boot: When enabled, the bootloader will be compiled with code to verify that an app is signed before booting it. If hardware secure boot is enabled, this option is always enabled and cannot be disabled. If hardware secure boot is not enabled, this option doesn't add significant security by itself so most users will want to leave it disabled.

How To Enable Signed App Verification

1. Open *Project Configuration Menu* -> Security features -> Enable *CONFIG_SECURE_SIGNED_APPS_NO_SECURE_BOOT*
2. “Bootloader verifies app signatures” can be enabled, which verifies app on boot.
3. By default, “Sign binaries during build” will be enabled on selecting “Require signed app images” option, which will sign binary files as a part of build process. The file named in “Secure boot private signing key” will be used to sign the image.
4. If you disable “Sign binaries during build” option then you'll have to enter path of a public key file used to verify signed images in “Secure boot public signature verification key”. In this case, private signing key should be generated by following instructions in [Generating Secure Boot Signing Key](#); public verification key and signed image should be generated by following instructions in [Remote Signing of Images](#).

5.25.13 Advanced Features

JTAG Debugging

By default, when Secure Boot is enabled then JTAG debugging is disabled via eFuse. The bootloader does this on first boot, at the same time it enables Secure Boot.

See [JTAG with Flash Encryption or Secure Boot](#) for more information about using JTAG Debugging with either Secure Boot or signed app verification enabled.

5.26 Secure Boot V2

Important: The references in this document are related to Secure Boot v2, the preferred scheme from ESP32-ECO3 onwards. (Refer to [Secure Boot](#) for ESP32)

Secure Boot V2 uses RSA based app and bootloader verification. This document can also be referred for signing apps with the RSA scheme without signing the bootloader.

5.26.1 Background

Secure Boot protects a device from running unsigned code (verification at time of load). A new RSA based secure boot verification scheme (Secure Boot V2) has been introduced for ESP32 ECO3 onwards.

- The software bootloader's RSA-PSS signature is verified by the Mask ROM and it is executed post successful verification.
- The verified software bootloader verifies the RSA-PSS signature of the application image before it is executed.

5.26.2 Advantages

- The RSA public key is stored on the device. The corresponding RSA private key is kept secret on a server and is never accessed by the device.
- Same image format & signature verification is applied for applications & software bootloader.
- No secrets are stored on the device. Therefore immune to passive side-channel attacks (timing or power analysis, etc.)

5.26.3 Secure Boot V2 Process

This is an overview of the Secure Boot V2 Process, Step by step instructions are supplied under [How To Enable Secure Boot V2](#).

1. Secure Boot V2 verifies the signature blocks appended to the bootloader and application binaries. The signature block contains the image binary signed by a RSA-3072 private key and its corresponding public key. More details on the [Signature Block Format](#).
2. On startup, ROM code checks the secure boot v2 bit in eFuse.
3. If secure boot is enabled, ROM checks the SHA-256 of the public key in the signature block in the eFuse.
4. The ROM code validates the public key embedded in the software bootloader's signature block by matching the SHA-256 of its public key to the SHA-256 in eFuse as per the earlier step. Boot process will be aborted if a valid hash of the public key isn't found in the eFuse.
5. The ROM code verifies the signature of the bootloader with the pre-validated public key with the RSA-PSS Scheme. In depth information on [Verifying the signature Block](#).
6. Software bootloader, reads the app partition and performs similar verification on the application. The application is verified on every boot up and OTA update. If selected OTA app partition fails verification, bootloader will fall back and look for another correctly signed partition.

5.26.4 Signature Block Format

The bootloader and application images are padded to the next 4096 byte boundary, thus the signature has a flash sector of its own. The signature is calculated over all bytes in the image including the padding bytes.

Each signature block contains the following:

- **Offset 0 (1 byte):** Magic byte (0xe7)
- **Offset 1 (1 byte):** Version number byte (currently 0x02), 0x01 is for Secure Boot V1.
- **Offset 2 (2 bytes):** Padding bytes, Reserved. Should be zero.
- **Offset 4 (32 bytes):** SHA-256 hash of only the image content, not including the signature block.
- **Offset 36 (384 bytes):** RSA Public Modulus used for signature verification. (value ‘n’ in RFC8017).
- **Offset 420 (4 bytes):** RSA Public Exponent used for signature verification (value ‘e’ in RFC8017).
- **Offset 424 (384 bytes):** Precalculated R, derived from ‘n’.
- **Offset 808 (4 bytes):** Precalculated M’, derived from ‘n’.
- **Offset 812 (384 bytes):** RSA-PSS Signature result (section 8.1.1 of RFC8017) of image content, computed using following PSS parameters: SHA256 hash, MFG1 function, 0 length salt, default trailer field (0xBC).
- **Offset 1196:** CRC32 of the preceding 1095 bytes.
- **Offset 1200 (16 bytes):** Zero padding to length 1216 bytes.

Note: R and M’ are used for hardware-assisted Montgomery Multiplication.

The remainder of the signature sector is erased flash (0xFF) which allows writing other signature blocks after previous signature block.

5.26.5 Verifying the signature Block

A signature block is “valid” if the first byte is 0xe7 and a valid CRC32 is stored at offset 1196.

An image is “verified” if the public key stored in any signature block is valid for this device, and if the stored signature is valid for the image data read from flash.

1. The magic byte, signature block CRC is validated.
 2. Public key digests are generated per signature block and compared with the digests from eFuse. If none of the digests match, the verification process is aborted.
 3. The application image digest is generated and matched with the image digest in the signature blocks. The verification process is aborted if the digests don’t match.
 4. The public key is used to verify the signature of the bootloader image, using RSA-PSS (section 8.1.2 of RFC8017) with the image digest calculated in step (3) for comparison.
- The application signing scheme is set to RSA for secure boot V2 and to ECDSA for secure boot V1.

Important: It is recommended to use secure boot V2 on the chip versions supporting them.

5.26.6 Bootloader Size

```
{IDF_TARGET_MAX_BOOTLOADER_SIZE:default = "64KB (0x10000 bytes)" , esp32 = "48KB (0xC000 bytes)" } {IDF_TARGET_MAX_PARTITION_TABLE_OFFSET:default = "0x12000" , esp32 = "0xE000" } ..
```

Above is calculated as 0x1000 at start of flash + IDF_TARGET_MAX_BOOTLOADER_SIZE + 0x1000 signature sector

When secure boot is enabled the bootloader app binary `bootloader.bin` may exceed the default bootloader size limit. This is especially likely if flash encryption is enabled as well. The default size limit is 0x7000 (28672) bytes (partition table offset 0x8000 - bootloader offset 0x1000).

If the bootloader becomes too large, the ESP32 will fail to boot - errors will be logged about either invalid partition table or invalid bootloader checksum.

When Secure Boot V2 is enabled, there is also an absolute binary size limit of {IDF_TARGET_MAX_BOOTLOADER_SIZE} (excluding the 4KB signature), because the bootloader is first loaded into a fixed size buffer for verification.

Options to work around this are:

- Set *bootloader compiler optimization* back to “Size” if it has been changed from this default value.
- Reduce *bootloader log level*. Setting log level to Warning, Error or None all significantly reduce the final binary size (but may make it harder to debug).
- Set *partition table offset* to a higher value than 0x8000, to place the partition table later in the flash. This increases the space available for the bootloader. If the *partition table* CSV file contains explicit partition offsets, they will need changing so no partition has an offset lower than CONFIG_PARTITION_TABLE_OFFSET + 0x1000. (This includes the default partition CSV files supplied with ESP-IDF.)
Note that because of the absolute binary size limit, there is no benefit to moving the partition table any higher than offset {IDF_TARGET_MAX_PARTITION_TABLE_OFFSET}.

5.26.7 eFuse usage

ESP32-ECO3:

- ABS_DONE_1 - Enables secure boot protection on boot.
- BLK2 - Stores the SHA-256 digest of the public key. SHA-256 hash of public key modulus, exponent, pre-calculated R & M’ values (represented as 776 bytes –offsets 36 to 812 - as per the *Signature Block Format*) is written to an eFuse key block.

5.26.8 How To Enable Secure Boot V2

1. Open the *Project Configuration Menu*, in “Security Features” set “Enable hardware Secure Boot in bootloader” to enable Secure Boot. The chip revision should be changed to revision 3(ESP32- ECO3) to view the Secure Boot V2 option.
2. To change the chip revision, set “Minimum Supported ESP32 Revision” to Rev 3 in “Component Config” -> “ESP32- Specific” , the Secure Boot V2 option can be enabled under “Enable hardware Secure Boot in bootloader” -> “Secure Boot Version” . Secure Boot V2 is available for ESP32 ECO3 onwards.
3. Specify the secure boot signing key path. The file can be anywhere on your system. A relative path will be evaluated from the project directory. The file does not need to exist yet.
4. Set other menuconfig options (as desired). Pay particular attention to the “Bootloader Config” options, as you can only flash the bootloader once. Then exit menuconfig and save your configuration
5. The first time you run `make` or `idf.py build`, if the signing key is not found then an error message will be printed with a command to generate a signing key via `espsecure.py generate_signing_key`.

Important: A signing key generated this way will use the best random number source available to the OS and its Python installation (`/dev/urandom` on OSX/Linux and `CryptGenRandom()` on Windows). If this random number source is weak, then the private key will be weak.

Important: For production environments, we recommend generating the keypair using openssl or another industry standard encryption program. See *Generating Secure Boot Signing Key* for more details.

6. Run `idf.py bootloader` to build a secure boot enabled bootloader. The build output will include a prompt for a flashing command, using `esptool.py write_flash`.
7. When you’ re ready to flash the bootloader, run the specified command (you have to enter it yourself, this step is not performed by the build system) and then wait for flashing to complete.
8. Run `idf.py flash` to build and flash the partition table and the just-built app image. The app image will be signed using the signing key you generated in step 4.

Note: `idf.py flash` doesn't flash the bootloader if secure boot is enabled.

9. Reset the ESP32 and it will boot the software bootloader you flashed. The software bootloader will enable secure boot on the chip, and then it verifies the app image signature and boots the app. You should watch the serial console output from the ESP32 to verify that secure boot is enabled and no errors have occurred due to the build configuration.

Note: Secure boot won't be enabled until after a valid partition table and app image have been flashed. This is to prevent accidents before the system is fully configured.

Note: If the ESP32 is reset or powered down during the first boot, it will start the process again on the next boot.

10. On subsequent boots, the secure boot hardware will verify the software bootloader has not changed and the software bootloader will verify the signed app image (using the validated public key portion of its appended signature block).

5.26.9 Restrictions after Secure Boot is enabled

- Any updated bootloader or app will need to be signed with a key matching the digest already stored in efuse.
- After Secure Boot is enabled, no further efuses can be read protected. (If *Flash Encryption* is enabled then the bootloader will ensure that any flash encryption key generated on first boot will already be read protected.) If `CONFIG_SECURE_BOOT_INSECURE` is enabled then this behaviour can be disabled, but this is not recommended.

5.26.10 Generating Secure Boot Signing Key

The build system will prompt you with a command to generate a new signing key via `espsecure.py generate_signing_key`. The `--version 2` parameter will generate the RSA 3072 private key for Secure Boot V2.

The strength of the signing key is proportional to (a) the random number source of the system, and (b) the correctness of the algorithm used. For production devices, we recommend generating signing keys from a system with a quality entropy source, and using the best available RSA key generation utilities.

For example, to generate a signing key using the `openssl` command line:

```
` openssl genrsa -out my_secure_boot_signing_key.pem 3072 `
```

Remember that the strength of the secure boot system depends on keeping the signing key private.

5.26.11 Remote Signing of Images

For production builds, it can be good practice to use a remote signing server rather than have the signing key on the build machine (which is the default `esp-idf` secure boot configuration). The `espsecure.py` command line program can be used to sign app images & partition table data for secure boot, on a remote system.

To use remote signing, disable the option “Sign binaries during build”. The private signing key does not need to be present on the build system.

After the app image and partition table are built, the build system will print signing steps using `espsecure.py`:

```
espsecure.py sign_data --version 2 --keyfile PRIVATE_SIGNING_KEY BINARY_FILE
```

The above command appends the image signature to the existing binary. You can use the `-output` argument to write the signed binary to a separate file:

```
espsecure.py sign_data --version 2 --keyfile PRIVATE_SIGNING_KEY --output SIGNED_  
↔BINARY_FILE BINARY_FILE
```

5.26.12 Secure Boot Best Practices

- Generate the signing key on a system with a quality source of entropy.
- Keep the signing key private at all times. A leak of this key will compromise the secure boot system.
- Do not allow any third party to observe any aspects of the key generation or signing process using `espsecure.py`. Both processes are vulnerable to timing or other side-channel attacks.
- Enable all secure boot options in the Secure Boot Configuration. These include flash encryption, disabling of JTAG, disabling BASIC ROM interpreter, and disabling the UART bootloader encrypted flash access.
- Use secure boot in combination with [flash encryption](#) to prevent local readout of the flash contents.

5.26.13 Technical Details

The following sections contain low-level reference descriptions of various secure boot elements:

Manual Commands

Secure boot is integrated into the `esp-idf` build system, so `make` or `idf.py build` will sign an app image and `idf.py bootloader` will produce a signed bootloader if secure signed binaries on build is enabled.

However, it is possible to use the `espsecure.py` tool to make standalone signatures and digests.

To sign a binary image:

```
espsecure.py sign_data --version 2 --keyfile ./my_signing_key.pem --output ./image_  
↔signed.bin image-unsigned.bin
```

Keyfile is the PEM file containing an RSA-3072 private signing key.

5.26.14 Secure Boot & Flash Encryption

If secure boot is used without [Flash Encryption](#), it is possible to launch “time-of-check to time-of-use” attack, where flash contents are swapped after the image is verified and running. Therefore, it is recommended to use both the features together.

5.26.15 Advanced Features

JTAG Debugging

By default, when Secure Boot is enabled then JTAG debugging is disabled via eFuse. The bootloader does this on first boot, at the same time it enables Secure Boot.

See [JTAG with Flash Encryption or Secure Boot](#) for more information about using JTAG Debugging with either Secure Boot or signed app verification enabled.

5.27 Thread Local Storage

5.27.1 Overview

Thread-local storage (TLS) is a mechanism by which variables are allocated such that there is one instance of the variable per extant thread. ESP-IDF provides three ways to make use of such variables:

- *FreeRTOS Native API*: ESP-IDF FreeRTOS native API.
- *Pthread API*: ESP-IDF's pthread API.
- *C11 Standard*: C11 standard introduces special keyword to declare variables as thread local.

5.27.2 FreeRTOS Native API

The ESP-IDF FreeRTOS provides the following API to manage thread local variables:

- `vTaskSetThreadLocalStoragePointer()`
- `pvTaskGetThreadLocalStoragePointer()`
- `vTaskSetThreadLocalStoragePointerAndDelCallback()`

In this case maximum number of variables that can be allocated is limited by `configNUM_THREAD_LOCAL_STORAGE_POINTERS` macro. Variables are kept in the task control block (TCB) and accessed by their index. Note that index 0 is reserved for ESP-IDF internal uses. Using that API user can allocate thread local variables of an arbitrary size and assign them to any number of tasks. Different tasks can have different sets of TLS variables. If size of the variable is more than 4 bytes then user is responsible for allocating/deallocating memory for it. Variable's deallocation is initiated by FreeRTOS when task is deleted, but user must provide function (callback) to do proper cleanup.

5.27.3 Pthread API

The ESP-IDF provides the following pthread API to manage thread local variables:

- `pthread_key_create()`
- `pthread_key_delete()`
- `pthread_getspecific()`
- `pthread_setspecific()`

This API has all benefits of the one above, but eliminates some its limits. The number of variables is limited only by size of available memory on the heap. Due to the dynamic nature this API introduces additional performance overhead compared to the native one.

5.27.4 C11 Standard

The ESP-IDF FreeRTOS supports thread local variables according to C11 standard (ones specified with `__thread` keyword). For details on this GCC feature please see <https://gcc.gnu.org/onlinedocs/gcc-5.5.0/gcc/Thread-Local.html#Thread-Local>. Storage for that kind of variables is allocated on the task's stack. Note that area for all such variables in the program will be allocated on the stack of every task in the system even if that task does not use such variables at all. For example ESP-IDF system tasks (like `ipc`, `timer` tasks etc.) will also have that extra stack space allocated. So this feature should be used with care. There is a tradeoff: C11 thread local variables are quite handy to use in programming and can be accessed using just a few Xtensa instructions, but this benefit goes with the cost of additional stack usage for all tasks in the system. Due to static nature of variables allocation all tasks in the system have the same sets of C11 thread local variables.

5.28 Tools

5.28.1 IDF Docker Image

IDF Docker image (`espressif/idf`) is intended for building applications and libraries with specific versions of ESP-IDF, when doing automated builds.

The image contains:

- Common utilities such as `git`, `wget`, `curl`, `zip`.
- Python 3.6 or newer.

- A copy of a specific version of ESP-IDF (see below for information about versions). `IDF_PATH` environment variable is set, and points to ESP-IDF location in the container.
- All the build tools required for the specific version of ESP-IDF: CMake, make, ninja, cross-compiler toolchains, etc.
- All Python packages required by ESP-IDF are installed in a virtual environment.

The image entrypoint sets up `PATH` environment variable to point to the correct version of tools, and activates the Python virtual environment. As a result, the environment is ready to use the ESP-IDF build system.

The image can also be used as a base for custom images, if additional utilities are required.

Tags

Multiple tags of this image are maintained:

- `latest`: tracks master branch of ESP-IDF
- `vX.Y`: corresponds to ESP-IDF release `vX.Y`
- `release-vX.Y`: tracks `release/vX.Y` branch of ESP-IDF

Note: Versions of ESP-IDF released before this feature was introduced do not have corresponding Docker image versions. You can check the up-to-date list of available tags at <https://hub.docker.com/r/ espressif/idf/tags>.

Usage

Setting up Docker Before using the `espressif/idf` Docker image locally, make sure you have Docker installed. Follow the instructions at <https://docs.docker.com/install/>, if it is not installed yet.

If using the image in CI environment, consult the documentation of your CI service on how to specify the image used for the build process.

Building a project with CMake In the project directory, run:

```
docker run --rm -v $PWD:/project -w /project espressif/idf idf.py build
```

The above command explained:

- `docker run`: runs a Docker image. It is a shorter form of the command `docker container run`.
- `--rm`: removes the container when the build is finished
- `-v $PWD:/project`: mounts the current directory on the host (`$PWD`) as `/project` directory in the container
- `espressif/idf`: uses Docker image `espressif/idf` with tag `latest` (implicitly added by Docker when no tag is specified)
- `idf.py build`: runs this command inside the container

To build with a specific docker image tag, specify it as `espressif/idf:TAG`:

```
docker run --rm -v $PWD:/project -w /project espressif/idf:v4.0 idf.py build
```

Note: At the time of writing, v4.0 release of ESP-IDF does not exist, yet, so the above command will not work. You can check the up-to-date list of available tags at <https://hub.docker.com/r/ espressif/idf/tags>.

Building a project with GNU Make Same as for CMake, except that the build command is different:

```
docker run --rm -v $PWD:/project -w /project espressif/idf make defconfig all -j4
```

Note: If the `sdkconfig` file does not exist, the default behavior of GNU Make build system is to open the menuconfig UI. This may be not desired in automated build environments. To ensure that the `sdkconfig` file exists, `defconfig` target is added before `all`.

Using the image interactively It is also possible to do builds interactively, to debug build issues or test the automated build scripts. Start the container with `-i -t` flags:

```
docker run --rm -v $PWD:/project -w /project -it espressif/idf
```

Then inside the container, use `idf.py` as usual:

```
idf.py menuconfig
idf.py build
```

Note: Commands which communicate with the development board, such as `idf.py flash` and `idf.py monitor` will not work in the container unless the serial port is passed through into the container. However currently this is not possible with Docker for Windows (<https://github.com/docker/for-win/issues/1018>) and Docker for Mac (<https://github.com/docker/for-mac/issues/900>).

5.28.2 IDF Component Manager

The IDF Component manager is a tool that downloads dependencies for any ESP-IDF CMake project. The download happens automatically during a run of CMake. It can source components either from [the component registry](#) or from a git repository.

A list of components can be found on <https://components.espressif.com/>

Activating the Component Manager

If CMake is started using `idf.py` or [ESP-IDF VSCode Extension](#) then the component manager will be activated by default.

If CMake is used directly or with some CMake-based IDE like CLion, it's necessary to set the `IDF_COMPONENT_MANAGER` environment variable to `1` to enable the component manager integration with the build system.

Using with a project

Dependencies for each component in the project are defined in a separate manifest file named `idf_component.yml` placed in the root of the component. The manifest file template can be created for a component by running `idf.py create-manifest --component=my_component`. When a new manifest is added to one of the components in the project it's necessary to reconfigure it manually by running `idf.py reconfigure`. Then build will track changes in `idf_component.yml` manifests and automatically triggers CMake when necessary.

There is an example application: `example:build_system/cmake/component_manager` that uses components installed by the component manager.

It's not necessary to have a manifest for components that don't need any managed dependencies.

When CMake configures the project (e.g. `idf.py reconfigure`) component manager does a few things:

- Processes `idf_component.yml` manifests for every component in the project and recursively solves dependencies
- Creates a `dependencies.lock` file in the root of the project with a full list of dependencies
- Downloads all dependencies to the `managed_components` directory

The lock-file `dependencies.lock` and content of `managed_components` directory is not supposed to be modified by a user. When the component manager runs it always make sure they are up to date. If these files were accidentally modified it's possible to re-run the component manager by triggering CMake with `idf.py reconfigure`

Defining dependencies in the manifest

```
dependencies:
  # Required IDF version
  idf: ">=4.1"
  # Defining a dependency from the registry:
  # https://components.espressif.com/component/example/cmp
  example/cmp: ">=1.0.0"

  # # Other ways to define dependencies
  #
  # # For components maintained by Espressif only name can be used.
  # # Same as `espressif/cmp`
  # component: "~1.0.0"
  #
  # # Or in a longer form with extra parameters
  # component2:
  #   version: ">=2.0.0"
  #
  # # For transient dependencies `public` flag can be set.
  # # `public` flag doesn't affect the `main` component.
  # # All dependencies of `main` are public by default.
  # public: true
  #
  # # For components hosted on non-default registry:
  # service_url: "https://componentregistry.company.com"
  #
  # # For components in git repository:
  # test_component:
  #   path: test_component
  #   git: ssh://git@gitlab.com/user/components.git
  #
  # # For test projects during component development
  # # components can be used from a local directory
  # # with relative or absolute path
  # some_local_component:
  #   path: ../../projects/component
```

5.29 ULP Coprocessor programming

5.29.1 ESP32 ULP coprocessor instruction set

This document provides details about the instructions used by ESP32 ULP coprocessor assembler.

ULP coprocessor has 4 16-bit general purpose registers, labeled R0, R1, R2, R3. It also has an 8-bit counter register (`stage_cnt`) which can be used to implement loops. Stage count register is accessed using special instructions.

ULP coprocessor can access 8k bytes of `RTC_SLOW_MEM` memory region. Memory is addressed in 32-bit word units. It can also access peripheral registers in `RTC_CNTL`, `RTC_IO`, and `SENS` peripherals.

All instructions are 32-bit. Jump instructions, ALU instructions, peripheral register and memory access instructions are executed in 1 cycle. Instructions which work with peripherals (TSENS, ADC, I2C) take variable number of cycles, depending on peripheral operation.

The instruction syntax is case insensitive. Upper and lower case letters can be used and intermixed arbitrarily. This is true both for register names and instruction names.

Note about addressing

ESP32 ULP coprocessor's `JUMP`, `ST`, `LD` instructions which take register as an argument (jump address, store/load base address) expect the argument to be expressed in 32-bit words.

Consider the following example program:

```
entry:
    NOP
    NOP
    NOP
    NOP
loop:
    MOVE R1, loop
    JUMP R1
```

When this program is assembled and linked, address of label `loop` will be equal to 16 (expressed in bytes). However `JUMP` instruction expects the address stored in register to be expressed in 32-bit words. To account for this common use case, assembler will convert the address of label `loop` from bytes to words, when generating `MOVE` instruction, so the code generated code will be equivalent to:

```
0000    NOP
0004    NOP
0008    NOP
000c    NOP
0010    MOVE R1, 4
0014    JUMP R1
```

The other case is when the argument of `MOVE` instruction is not a label but a constant. In this case assembler will use the value as is, without any conversion:

```
.set     val, 0x10
MOVE     R1, val
```

In this case, value loaded into R1 will be 0x10.

Similar considerations apply to `LD` and `ST` instructions. Consider the following code:

```
.global array
array: .long 0
       .long 0
       .long 0
       .long 0

MOVE R1, array
MOVE R2, 0x1234
ST R2, R1, 0    // write value of R2 into the first array element,
                // i.e. array[0]

ST R2, R1, 4    // write value of R2 into the second array element
                // (4 byte offset), i.e. array[1]

ADD R1, R1, 2   // this increments address by 2 words (8 bytes)
ST R2, R1, 0    // write value of R2 into the third array element,
                // i.e. array[2]
```

Note about instruction execution time

ULP coprocessor is clocked from RTC_FAST_CLK, which is normally derived from the internal 8MHz oscillator. Applications which need to know exact ULP clock frequency can calibrate it against the main XTAL clock:

```
#include "soc/rtc.h"

// calibrate 8M/256 clock against XTAL, get 8M/256 clock period
uint32_t rtc_8md256_period = rtc_clk_cal(RTC_CAL_8MD256, 100);
uint32_t rtc_fast_freq_hz = 1000000ULL * (1 << RTC_CLK_CAL_FRACT) * 256 / rtc_
↪8md256_period;
```

ULP coprocessor needs certain number of clock cycles to fetch each instruction, plus certain number of cycles to execute it, depending on the instruction. See description of each instruction below for details on the execution time.

Instruction fetch time is:

- 2 clock cycles —for instructions following ALU and branch instructions.
- 4 clock cycles —in other cases.

Note that when accessing RTC memories and RTC registers, ULP coprocessor has lower priority than the main CPUs. This means that ULP coprocessor execution may be suspended while the main CPUs access same memory region as the ULP.

NOP - no operation**Syntax NOP**

Operands None

Cycles 2 cycle to execute, 4 cycles to fetch next instruction

Description No operation is performed. Only the PC is incremented.

Example:

```
1:    NOP
```

ADD - Add to register

Syntax **ADD** *Rdst, Rsrc1, Rsrc2*

ADD *Rdst, Rsrc1, imm*

Operands

- *Rdst* - Register R[0..3]
- *Rsrc1* - Register R[0..3]
- *Rsrc2* - Register R[0..3]
- *Imm* - 16-bit signed value

Cycles 2 cycles to execute, 4 cycles to fetch next instruction

Description The instruction adds source register to another source register or to a 16-bit signed value and stores result to the destination register.

Examples:

```
1:    ADD R1, R2, R3        //R1 = R2 + R3
2:    Add R1, R2, 0x1234   //R1 = R2 + 0x1234
3:    .set value1, 0x03    //constant value1=0x03
    Add R1, R2, value1    //R1 = R2 + value1
4:    .global label       //declaration of variable label
    Add R1, R2, label     //R1 = R2 + label
```

(continues on next page)

```

...
label: nop                //definition of variable label

```

SUB - Subtract from register

Syntax SUB *Rdst*, *Rsrc1*, *Rsrc2*

SUB *Rdst*, *Rsrc1*, *imm*

Operands

- *Rdst* - Register R[0..3]
- *Rsrc1* - Register R[0..3]
- *Rsrc2* - Register R[0..3]
- *Imm* - 16-bit signed value

Cycles 2 cycles to execute, 4 cycles to fetch next instruction

Description The instruction subtracts the source register from another source register or subtracts 16-bit signed value from a source register, and stores result to the destination register.

Examples:

```

1:      SUB R1, R2, R3           //R1 = R2 - R3
2:      sub R1, R2, 0x1234      //R1 = R2 - 0x1234
3:      .set value1, 0x03       //constant value1=0x03
      SUB R1, R2, value1        //R1 = R2 - value1
4:      .global label           //declaration of variable label
      SUB R1, R2, label         //R1 = R2 - label
      ...
label:  nop                     //definition of variable label

```

AND - Logical AND of two operands

Syntax AND *Rdst*, *Rsrc1*, *Rsrc2*

AND *Rdst*, *Rsrc1*, *imm*

Operands

- *Rdst* - Register R[0..3]
- *Rsrc1* - Register R[0..3]
- *Rsrc2* - Register R[0..3]
- *Imm* - 16-bit signed value

Cycles 2 cycles to execute, 4 cycles to fetch next instruction

Description The instruction does logical AND of a source register and another source register or 16-bit signed value and stores result to the destination register.

Examples:

```

1:      AND R1, R2, R3          //R1 = R2 & R3
2:      AND R1, R2, 0x1234     //R1 = R2 & 0x1234
3:      .set value1, 0x03      //constant value1=0x03
      AND R1, R2, value1       //R1 = R2 & value1
4:      .global label          //declaration of variable label
      AND R1, R2, label        //R1 = R2 & label
      ...
label:  nop                     //definition of variable label

```

OR - Logical OR of two operands**Syntax** `OR Rdst, Rsrc1, Rsrc2``OR Rdst, Rsrc1, imm`**Operands**

- *Rdst* - Register R[0..3]
- *Rsrc1* - Register R[0..3]
- *Rsrc2* - Register R[0..3]
- *Imm* - 16-bit signed value

Cycles 2 cycles to execute, 4 cycles to fetch next instruction**Description** The instruction does logical OR of a source register and another source register or 16-bit signed value and stores result to the destination register.**Examples:**

```

1:      OR R1, R2, R3           //R1 = R2 \| R3
2:      OR R1, R2, 0x1234      //R1 = R2 \| 0x1234
3:      .set value1, 0x03      //constant value1=0x03
        OR R1, R2, value1      //R1 = R2 \| value1
4:      .global label          //declaration of variable label
        OR R1, R2, label       //R1 = R2 \|label
        ...
label:  nop                    //definition of variable label

```

LSH - Logical Shift Left**Syntax** `LSH Rdst, Rsrc1, Rsrc2``LSH Rdst, Rsrc1, imm`**Operands**

- *Rdst* - Register R[0..3]
- *Rsrc1* - Register R[0..3]
- *Rsrc2* - Register R[0..3]
- *Imm* - 16-bit signed value

Cycles 2 cycles to execute, 4 cycles to fetch next instruction**Description** The instruction does logical shift to left of source register to number of bits from another source register or 16-bit signed value and store result to the destination register.**Examples:**

```

1:      LSH R1, R2, R3          //R1 = R2 << R3
2:      LSH R1, R2, 0x03       //R1 = R2 << 0x03
3:      .set value1, 0x03      //constant value1=0x03
        LSH R1, R2, value1     //R1 = R2 << value1
4:      .global label          //declaration of variable label
        LSH R1, R2, label      //R1 = R2 << label
        ...
label:  nop                    //definition of variable label

```

RSH - Logical Shift Right**Syntax** `RSH Rdst, Rsrc1, Rsrc2``RSH Rdst, Rsrc1, imm`**Operands** *Rdst* - Register R[0..3] *Rsrc1* - Register R[0..3] *Rsrc2* - Register R[0..3] *Imm* - 16-bit signed value

Cycles 2 cycles to execute, 4 cycles to fetch next instruction

Description The instruction does logical shift to right of source register to number of bits from another source register or 16-bit signed value and store result to the destination register.

Examples:

```

1:      RSH R1, R2, R3           //R1 = R2 >> R3
2:      RSH R1, R2, 0x03        //R1 = R2 >> 0x03
3:      .set value1, 0x03       //constant value1=0x03
      RSH R1, R2, value1        //R1 = R2 >> value1
4:      .global label           //declaration of variable label
      RSH R1, R2, label         //R1 = R2 >> label
label:  nop                     //definition of variable label

```

MOVE –Move to register

Syntax `MOVE Rdst, Rsrc`

`MOVE Rdst, imm`

Operands

- *Rdst* –Register R[0..3]
- *Rsrc* –Register R[0..3]
- *Imm* –16-bit signed value

Cycles 2 cycles to execute, 4 cycles to fetch next instruction

Description The instruction move to destination register value from source register or 16-bit signed value.

Note that when a label is used as an immediate, the address of the label will be converted from bytes to words.

This is because LD, ST, and JUMP instructions expect the address register value to be expressed in words rather than bytes. To avoid using an extra instruction

Examples:

```

1:      MOVE      R1, R2           //R1 = R2
2:      MOVE      R1, 0x03        //R1 = 0x03
3:      .set      value1, 0x03    //constant value1=0x03
      MOVE      R1, value1        //R1 = value1
4:      .global   label           //declaration of label
      MOVE      R1, label         //R1 = address_of(label) / 4
      ...
label:  nop                     //definition of label

```

ST –Store data to the memory

Syntax `ST Rsrc, Rdst, offset`

Operands

- *Rsrc* –Register R[0..3], holds the 16-bit value to store
- *Rdst* –Register R[0..3], address of the destination, in 32-bit words
- *Offset* –10-bit signed value, offset in bytes

Cycles 4 cycles to execute, 4 cycles to fetch next instruction

Description The instruction stores the 16-bit value of Rsrc to the lower half-word of memory with address Rdst+offset. The upper half-word is written with the current program counter (PC), expressed in words, shifted left by 5 bits:

```
Mem[Rdst + offset / 4]{31:0} = {PC[10:0], 5'b0, Rsrc[15:0]}
```

The application can use higher 16 bits to determine which instruction in the ULP program has written any particular word into memory.

Examples:

```

1:      ST  R1, R2, 0x12      //MEM[R2+0x12] = R1

2:      .data                //Data section definition
Addr1:  .word    123         // Define label Addr1 16 bit
        .set     offs, 0x00  // Define constant offs
        .text                //Text section definition
        MOVE    R1, 1       // R1 = 1
        MOVE    R2, Addr1   // R2 = Addr1
        ST      R1, R2, offs // MEM[R2 + 0] = R1
                               // MEM[Addr1 + 0] will be 32'h600001

```

LD –Load data from the memory

Syntax LD *Rdst, Rsrc, offset*

Operands *Rdst* –Register R[0..3], destination

Rsrc –Register R[0..3], holds address of destination, in 32-bit words

Offset –10-bit signed value, offset in bytes

Cycles 4 cycles to execute, 4 cycles to fetch next instruction

Description The instruction loads lower 16-bit half-word from memory with address *Rsrc*+*offset* into the destination register *Rdst*:

```
Rdst[15:0] = Mem[Rsrc + offset / 4][15:0]
```

Examples:

```

1:      LD  R1, R2, 0x12      //R1 = MEM[R2+0x12]

2:      .data                //Data section definition
Addr1:  .word    123         // Define label Addr1 16 bit
        .set     offs, 0x00  // Define constant offs
        .text                //Text section definition
        MOVE    R1, 1       // R1 = 1
        MOVE    R2, Addr1   // R2 = Addr1 / 4 (address of label is_
↳converted into words)
        LD      R1, R2, offs // R1 = MEM[R2 + 0]
                               // R1 will be 123

```

JUMP –Jump to an absolute address

Syntax JUMP *Rdst*

JUMP *ImmAddr*

JUMP *Rdst, Condition*

JUMP *ImmAddr, Condition*

Operands

- *Rdst* –Register R[0..3] containing address to jump to (expressed in 32-bit words)
- *ImmAddr* –13 bits address (expressed in bytes), aligned to 4 bytes
- **Condition:**
 - EQ –jump if last ALU operation result was zero
 - OV –jump if last ALU has set overflow flag

Cycles 2 cycles to execute, 2 cycles to fetch next instruction

Description The instruction makes jump to the specified address. Jump can be either unconditional or based on an ALU flag.

Examples:

```

1:      JUMP      R1          // Jump to address in R1 (address in R1 is in_
↳32-bit words)

2:      JUMP      0x120, EQ   // Jump to address 0x120 (in bytes) if ALU_
↳result is zero

3:      JUMP      label      // Jump to label
      ...
label:  nop              // Definition of label

4:      .global   label      // Declaration of global label

      MOVE      R1, label    // R1 = label (value loaded into R1 is in words)
      JUMP      R1          // Jump to label
      ...
label:  nop              // Definition of label

```

JUMPR – Jump to a relative offset (condition based on R0)

Syntax JUMPR *Step, Threshold, Condition*

Operands

- *Step* –relative shift from current position, in bytes
- *Threshold* –threshold value for branch condition
- **Condition:**
 - *EQ* (equal) –jump if value in R0 == threshold
 - *LT* (less than) –jump if value in R0 < threshold
 - *LE* (less or equal) –jump if value in R0 <= threshold
 - *GT* (greater than) –jump if value in R0 > threshold
 - *GE* (greater or equal) –jump if value in R0 >= threshold

Cycles Conditions *LT*, *GE*, *LE* and *GT*: 2 cycles to execute, 2 cycles to fetch next instruction

Conditions *LE* and *GT* are implemented in the assembler using one **JUMPR** instructions:

```

// JUMPR target, threshold, GT is implemented as:

      JUMPR target, threshold+1, GE

// JUMPR target, threshold, LE is implemented as:

      JUMPR target, threshold + 1, LT

```

Conditions *EQ* is implemented in the assembler using two **JUMPR** instructions:

```

// JUMPR target, threshold, EQ is implemented as:

      JUMPR next, threshold + 1, GE
      JUMPR target, threshold, GE
next:

```

Therefore the execution time will depend on the branches taken: either 2 cycles to execute + 2 cycles to fetch, or 4 cycles to execute + 4 cycles to fetch.

Description The instruction makes a jump to a relative address if condition is true. Condition is the result of comparison of R0 register value and the threshold value.

Examples:

```

1:pos:  JUMPR     16, 20, GE   // Jump to address (position + 16 bytes) if_
↳value in R0 >= 20

2:      // Down counting loop using R0 register
      MOVE      R0, 16       // load 16 into R0

```

(continues on next page)

```

label:  SUB      R0, R0, 1    // R0--
        NOP                      // do something
        JUMPR   label, 1, GE // jump to label if R0 >= 1

```

JUMPS –Jump to a relative address (condition based on stage count)

Syntax JUMPS *Step, Threshold, Condition*

Operands

- *Step* –relative shift from current position, in bytes
- *Threshold* –threshold value for branch condition
- **Condition:**
 - *EQ* (equal) –jump if value in stage_cnt == threshold
 - *LT* (less than) –jump if value in stage_cnt < threshold
 - *LE* (less or equal) - jump if value in stage_cnt <= threshold
 - *GT* (greater than) –jump if value in stage_cnt > threshold
 - *GE* (greater or equal) —jump if value in stage_cnt >= threshold

Cycles Conditions *LE, LT, GE*: 2 cycles to execute, 2 cycles to fetch next instruction

Conditions *EQ, GT* are implemented in the assembler using two **JUMPS** instructions:

```

// JUMPS target, threshold, EQ is implemented as:

        JUMPS next, threshold, LT
        JUMPS target, threshold, LE
next:

// JUMPS target, threshold, GT is implemented as:

        JUMPS next, threshold, LE
        JUMPS target, threshold, GE
next:

```

Therefore the execution time will depend on the branches taken: either 2 cycles to execute + 2 cycles to fetch, or 4 cycles to execute + 4 cycles to fetch.

Description The instruction makes a jump to a relative address if condition is true. Condition is the result of comparison of count register value and threshold value.

Examples:

```

1:pos:  JUMPS    16, 20, EQ    // Jump to (position + 16 bytes) if stage_cnt_
↳ == 20

2:      // Up counting loop using stage count register
        STAGE_RST          // set stage_cnt to 0
label:  STAGE_INC 1          // stage_cnt++
        NOP                // do something
        JUMPS    label, 16, LT // jump to label if stage_cnt < 16

```

STAGE_RST –Reset stage count register

Syntax STAGE_RST

Operands No operands

Description The instruction sets the stage count register to 0

Cycles 2 cycles to execute, 4 cycles to fetch next instruction

Examples:

```

1:      STAGE_RST          // Reset stage count register

```

STAGE_INC –Increment stage count register**Syntax** STAGE_INC *Value***Operands**

- *Value* –8 bits value

Cycles 2 cycles to execute, 4 cycles to fetch next instruction

Description The instruction increments stage count register by given value.

Examples:

```

1:      STAGE_INC      10          // stage_cnt += 10

2:      // Up counting loop example:
        STAGE_RST          // set stage_cnt to 0
label:  STAGE_INC      1          // stage_cnt++
        NOP                // do something
        JUMPS          label, 16, LT // jump to label if stage_cnt < 16

```

STAGE_DEC –Decrement stage count register**Syntax** STAGE_DEC *Value***Operands**

- *Value* –8 bits value

Cycles 2 cycles to execute, 4 cycles to fetch next instruction

Description The instruction decrements stage count register by given value.

Examples:

```

1:      STAGE_DEC      10          // stage_cnt -= 10;

2:      // Down counting loop example
        STAGE_RST          // set stage_cnt to 0
        STAGE_INC      16          // increment stage_cnt to 16
label:  STAGE_DEC      1          // stage_cnt--;
        NOP                // do something
        JUMPS          label, 0, GT // jump to label if stage_cnt > 0

```

HALT –End the program**Syntax** HALT

Operands No operands

Cycles 2 cycles to execute

Description The instruction halts the ULP coprocessor and restarts ULP wakeup timer, if it is enabled.

Examples:

```

1:      HALT          // Halt the coprocessor

```

WAKE –Wake up the chip**Syntax** WAKE

Operands No operands

Cycles 2 cycles to execute, 4 cycles to fetch next instruction

Description The instruction sends an interrupt from ULP to RTC controller.

- If the SoC is in deep sleep mode, and ULP wakeup is enabled, this causes the SoC to wake up.
- If the SoC is not in deep sleep mode, and ULP interrupt bit (RTC_CNTL_ULP_CP_INT_ENA) is set in RTC_CNTL_INT_ENA_REG register, RTC interrupt will be triggered.

Note that before using WAKE instruction, ULP program may needs to wait until RTC controller is ready to wake up the main CPU. This is indicated using RTC_CNTL_RDY_FOR_WAKEUP bit of RTC_CNTL_LOW_POWER_ST_REG register. If WAKE instruction is executed while RTC_CNTL_RDY_FOR_WAKEUP is zero, it has no effect (wake up does not occur).

Examples:

```

1: is_rdy_for_wakeup:                // Read RTC_CNTL_RDY_FOR_WAKEUP bit
    READ_RTC_FIELD(RTC_CNTL_LOW_POWER_ST_REG, RTC_CNTL_RDY_FOR_WAKEUP)
    AND r0, r0, 1
    JUMP is_rdy_for_wakeup, eq      // Retry until the bit is set
    WAKE                            // Trigger wake up
    REG_WR 0x006, 24, 24, 0        // Stop ULP timer (clear RTC_CNTL_ULP_CP_
↳SLP_TIMER_EN)
    HALT                            // Stop the ULP program
    // After these instructions, SoC will wake up,
    // and ULP will not run again until started by the main program.

```

SLEEP –set ULP wakeup timer period

Syntax SLEEP *sleep_reg*

Operands

- *sleep_reg* –0..4, selects one of SENS_ULP_CP_SLEEP_CYC_x_REG registers.

Cycles 2 cycles to execute, 4 cycles to fetch next instruction

Description The instruction selects which of the SENS_ULP_CP_SLEEP_CYC_x_REG (x = 0..4) register values is to be used by the ULP wakeup timer as wakeup period. By default, the value from SENS_ULP_CP_SLEEP_CYC0_REG is used.

Examples:

```

1:      SLEEP    1          // Use period set in SENS_ULP_CP_SLEEP_CYC1_REG

2:      .set sleep_reg, 4  // Set constant
        SLEEP  sleep_reg  // Use period set in SENS_ULP_CP_SLEEP_CYC4_REG

```

WAIT –wait some number of cycles

Syntax WAIT *Cycles*

Operands

- *Cycles* –number of cycles for wait

Cycles 2 + *Cycles* cycles to execute, 4 cycles to fetch next instruction

Description The instruction delays for given number of cycles.

Examples:

```

1:      WAIT     10         // Do nothing for 10 cycles

2:      .set  wait_cnt, 10  // Set a constant
        WAIT  wait_cnt     // wait for 10 cycles

```

TSENS –do measurement with temperature sensor

Syntax

- TSENS *Rdst*, *Wait_Delay*

Operands

- *Rdst* –Destination Register R[0..3], result will be stored to this register
- *Wait_Delay* –number of cycles used to perform the measurement

Cycles 2 + *Wait_Delay* + 3 * TSENS_CLK to execute, 4 cycles to fetch next instruction

Description The instruction performs measurement using TSENS and stores the result into a general purpose register.

Examples:

```
1:      TSENS      R1, 1000      // Measure temperature sensor for 1000 cycles,
                                     // and store result to R1
```

ADC –do measurement with ADC

Syntax

- **ADC** *Rdst, Sar_sel, Mux*
- **ADC** *Rdst, Sar_sel, Mux, 0* —deprecated form

Operands

- *Rdst* –Destination Register R[0..3], result will be stored to this register
- *Sar_sel* –Select ADC: 0 = SARADC1, 1 = SARADC2
- *Mux* - Enable ADC channel. Channel number is [Mux-1]. If the user passes Mux value 1, then ADC channel 0 gets used.

Cycles $23 + \max(1, \text{SAR_AMP_WAIT1}) + \max(1, \text{SAR_AMP_WAIT2}) + \max(1, \text{SAR_AMP_WAIT3}) + \text{SARx_SAMPLE_CYCLE} + \text{SARx_SAMPLE_BIT}$ cycles to execute, 4 cycles to fetch next instruction

Description The instruction makes measurements from ADC.

Examples:

```
1:      ADC        R1, 0, 1      // Measure value using ADC1 channel 0 and store_
↳result into R1
```

I2C_RD - read single byte from I2C slave

Syntax

- **I2C_RD** *Sub_addr, High, Low, Slave_sel*

Operands

- *Sub_addr* –Address within the I2C slave to read.
- *High, Low* —Define range of bits to read. Bits outside of [High, Low] range are masked.
- *Slave_sel* - Index of I2C slave address to use.

Cycles Execution time mostly depends on I2C communication time. 4 cycles to fetch next instruction.

Description I2C_RD instruction reads one byte from I2C slave with index *Slave_sel*. Slave address (in 7-bit format) has to be set in advance into *SENS_I2C_SLAVE_ADDRx* register field, where $x == \text{Slave_sel}$. 8 bits of read result is stored into *R0* register.

Examples:

```
1:      I2C_RD     0x10, 7, 0, 0  // Read byte from sub-address 0x10 of_
↳slave with address set in SENS_I2C_SLAVE_ADDR0
```

I2C_WR - write single byte to I2C slave

Syntax

- **I2C_WR** *Sub_addr, Value, High, Low, Slave_sel*

Operands

- *Sub_addr* –Address within the I2C slave to write.
- *Value* –8-bit value to be written.
- *High, Low* —Define range of bits to write. Bits outside of [High, Low] range are masked.
- *Slave_sel* - Index of I2C slave address to use.

Cycles Execution time mostly depends on I2C communication time. 4 cycles to fetch next instruction.

Description I2C_WR instruction writes one byte to I2C slave with index `Slave_sel`. Slave address (in 7-bit format) has to be set in advance into `SENS_I2C_SLAVE_ADDRx` register field, where `x == Slave_sel`.

Examples:

```
1:          I2C_WR          0x20, 0x33, 7, 0, 1          // Write byte 0x33 to sub-address_
↪0x20 of slave with address set in SENS_I2C_SLAVE_ADDR1.
```

REG_RD –read from peripheral register

Syntax REG_RD *Addr, High, Low*

Operands

- *Addr* –Register address, in 32-bit words
- *High* –Register end bit number
- *Low* –Register start bit number

Cycles 4 cycles to execute, 4 cycles to fetch next instruction

Description The instruction reads up to 16 bits from a peripheral register into a general purpose register: `R0 = REG[Addr][High:Low]`.

This instruction can access registers in `RTC_CNTL`, `RTC_IO`, `SENS`, and `RTC_I2C` peripherals. Address of the the register, as seen from the ULP, can be calculated from the address of the same register on the DPORT bus as follows:

$$\text{addr_ulp} = (\text{addr_dport} - \text{DR_REG_RTCCNTL_BASE}) / 4$$

Examples:

```
1:          REG_RD          0x120, 7, 4          // load 4 bits: R0 = {12'b0, REG[0x120][7:4]}
```

REG_WR –write to peripheral register

Syntax REG_WR *Addr, High, Low, Data*

Operands

- *Addr* –Register address, in 32-bit words.
- *High* –Register end bit number
- *Low* –Register start bit number
- *Data* –Value to write, 8 bits

Cycles 8 cycles to execute, 4 cycles to fetch next instruction

Description The instruction writes up to 8 bits from an immediate data value into a peripheral register: `REG[Addr][High:Low] = data`.

This instruction can access registers in `RTC_CNTL`, `RTC_IO`, `SENS`, and `RTC_I2C` peripherals. Address of the the register, as seen from the ULP, can be calculated from the address of the same register on the DPORT bus as follows:

$$\text{addr_ulp} = (\text{addr_dport} - \text{DR_REG_RTCCNTL_BASE}) / 4$$

Examples:

```
1:          REG_WR          0x120, 7, 0, 0x10          // set 8 bits: REG[0x120][7:0] = 0x10
```

Convenience macros for peripheral registers access

ULP source files are passed through C preprocessor before the assembler. This allows certain macros to be used to facilitate access to peripheral registers.

Some existing macros are defined in `soc/soc_ulp.h` header file. These macros allow access to the fields of peripheral registers by their names. Peripheral registers names which can be used with these macros are the ones

defined in `soc/rtc_cntl_reg.h`, `soc/rtc_io_reg.h`, `soc/sens_reg.h`, and `soc/rtc_i2c_reg.h`.

READ_RTC_REG(`rtc_reg`, `low_bit`, `bit_width`) Read up to 16 bits from `rtc_reg[low_bit + bit_width - 1 : low_bit]` into R0. For example:

```
#include "soc/soc_ulp.h"
#include "soc/rtc_cntl_reg.h"

/* Read 16 lower bits of RTC_CNTL_TIME0_REG into R0 */
READ_RTC_REG(RTC_CNTL_TIME0_REG, 0, 16)
```

READ_RTC_FIELD(`rtc_reg`, `field`) Read from a field in `rtc_reg` into R0, up to 16 bits. For example:

```
#include "soc/soc_ulp.h"
#include "soc/sens_reg.h"

/* Read 8-bit SENS_TSENS_OUT field of SENS_SAR_SLAVE_ADDR3_REG into R0 */
READ_RTC_FIELD(SENS_SAR_SLAVE_ADDR3_REG, SENS_TSENS_OUT)
```

WRITE_RTC_REG(`rtc_reg`, `low_bit`, `bit_width`, `value`) Write immediate value into `rtc_reg[low_bit + bit_width - 1 : low_bit]`, `bit_width <= 8`. For example:

```
#include "soc/soc_ulp.h"
#include "soc/rtc_io_reg.h"

/* Set BIT(2) of RTC_GPIO_OUT_DATA_W1TS field in RTC_GPIO_OUT_W1TS_REG */
WRITE_RTC_REG(RTC_GPIO_OUT_W1TS_REG, RTC_GPIO_OUT_DATA_W1TS_S + 2, 1, 1)
```

WRITE_RTC_FIELD(`rtc_reg`, `field`, `value`) Write immediate value into a field in `rtc_reg`, up to 8 bits. For example:

```
#include "soc/soc_ulp.h"
#include "soc/rtc_cntl_reg.h"

/* Set RTC_CNTL_ULP_CP_SLP_TIMER_EN field of RTC_CNTL_STATE0_REG to 0 */
WRITE_RTC_FIELD(RTC_CNTL_STATE0_REG, RTC_CNTL_ULP_CP_SLP_TIMER_EN, 0)
```

5.29.2 ESP32-S2 ULP coprocessor instruction set

This document provides details about the instructions used by ESP32-S2 ULP coprocessor assembler.

ULP coprocessor has 4 16-bit general purpose registers, labeled R0, R1, R2, R3. It also has an 8-bit counter register (`stage_cnt`) which can be used to implement loops. Stage count register is accessed using special instructions.

ULP coprocessor can access 8k bytes of `RTC_SLOW_MEM` memory region. Memory is addressed in 32-bit word units. It can also access peripheral registers in `RTC_CNTL`, `RTC_IO`, and `SENS` peripherals.

All instructions are 32-bit. Jump instructions, ALU instructions, peripheral register and memory access instructions are executed in 1 cycle. Instructions which work with peripherals (TSENS, ADC, I2C) take variable number of cycles, depending on peripheral operation.

The instruction syntax is case insensitive. Upper and lower case letters can be used and intermixed arbitrarily. This is true both for register names and instruction names.

Note about addressing

ESP32-S2 ULP coprocessor's JUMP, ST, LD instructions which take register as an argument (jump address, store/load base address) expect the argument to be expressed in 32-bit words.

Consider the following example program:

```

entry:
    NOP
    NOP
    NOP
    NOP
loop:
    MOVE R1, loop
    JUMP R1

```

When this program is assembled and linked, address of label `loop` will be equal to 16 (expressed in bytes). However `JUMP` instruction expects the address stored in register to be expressed in 32-bit words. To account for this common use case, assembler will convert the address of label `loop` from bytes to words, when generating `MOVE` instruction, so the code generated code will be equivalent to:

```

0000    NOP
0004    NOP
0008    NOP
000c    NOP
0010    MOVE R1, 4
0014    JUMP R1

```

The other case is when the argument of `MOVE` instruction is not a label but a constant. In this case assembler will use the value as is, without any conversion:

```

.set      val, 0x10
MOVE     R1, val

```

In this case, value loaded into R1 will be 0x10.

Similar considerations apply to `LD` and `ST` instructions. Consider the following code:

```

.global array
array: .long 0
       .long 0
       .long 0
       .long 0

MOVE R1, array
MOVE R2, 0x1234
ST R2, R1, 0 // write value of R2 into the first array element,
              // i.e. array[0]

ST R2, R1, 4 // write value of R2 into the second array element
              // (4 byte offset), i.e. array[1]

ADD R1, R1, 2 // this increments address by 2 words (8 bytes)
ST R2, R1, 0 // write value of R2 into the third array element,
              // i.e. array[2]

```

Note about instruction execution time

ULP coprocessor is clocked from `RTC_FAST_CLK`, which is normally derived from the internal 8MHz oscillator. Applications which need to know exact ULP clock frequency can calibrate it against the main XTAL clock:

```

#include "soc/rtc.h"

// calibrate 8M/256 clock against XTAL, get 8M/256 clock period
uint32_t rtc_8md256_period = rtc_clk_cal(RTC_CAL_8MD256, 100);
uint32_t rtc_fast_freq_hz = 1000000ULL * (1 << RTC_CLK_CAL_FRAC) * 256 / rtc_
↪8md256_period;

```

ULP coprocessor needs certain number of clock cycles to fetch each instruction, plus certain number of cycles to execute it, depending on the instruction. See description of each instruction below for details on the execution time.

Instruction fetch time is:

- 2 clock cycles —for instructions following ALU and branch instructions.
- 4 clock cycles —in other cases.

Note that when accessing RTC memories and RTC registers, ULP coprocessor has lower priority than the main CPUs. This means that ULP coprocessor execution may be suspended while the main CPUs access same memory region as the ULP.

Difference between ESP32 ULP and ESP32-S2 ULP Instruction sets

Compare to the ESP32 ULP coprocessor, the ESP-S2 ULP coprocessor has extended instruction set. The ESP32-S2 ULP is not binary compatible with ESP32 ULP, but the assembled program that was written for the ESP32 ULP will also work on the ESP32-S2 ULP after rebuild. The list of the new instructions that was added to the ESP32-S2 ULP is: LDL, LDH, STO, ST32, STI32. The detailed description of these commands please see below.

NOP - no operation

Syntax NOP

Operands None

Cycles 2 cycle to execute, 4 cycles to fetch next instruction

Description No operation is performed. Only the PC is incremented.

Example:

```
1:    NOP
```

ADD - Add to register

Syntax ADD *Rdst*, *Rsrc1*, *Rsrc2*

ADD *Rdst*, *Rsrc1*, *imm*

Operands

- *Rdst* - Register R[0..3]
- *Rsrc1* - Register R[0..3]
- *Rsrc2* - Register R[0..3]
- *Imm* - 16-bit signed value

Cycles 2 cycles to execute, 4 cycles to fetch next instruction

Description The instruction adds source register to another source register or to a 16-bit signed value and stores result to the destination register.

Examples:

```
1:    ADD R1, R2, R3           //R1 = R2 + R3
2:    Add R1, R2, 0x1234      //R1 = R2 + 0x1234
3:    .set value1, 0x03       //constant value1=0x03
    Add R1, R2, value1        //R1 = R2 + value1
4:    .global label           //declaration of variable label
    Add R1, R2, label         //R1 = R2 + label
    ...
    label: nop                //definition of variable label
```

SUB - Subtract from register**Syntax** SUB *Rdst, Rsrc1, Rsrc2*SUB *Rdst, Rsrc1, imm***Operands**

- *Rdst* - Register R[0..3]
- *Rsrc1* - Register R[0..3]
- *Rsrc2* - Register R[0..3]
- *Imm* - 16-bit signed value

Cycles 2 cycles to execute, 4 cycles to fetch next instruction**Description** The instruction subtracts the source register from another source register or subtracts 16-bit signed value from a source register, and stores result to the destination register.**Examples:**

```

1:      SUB R1, R2, R3           //R1 = R2 - R3
2:      sub R1, R2, 0x1234      //R1 = R2 - 0x1234
3:      .set value1, 0x03       //constant value1=0x03
      SUB R1, R2, value1        //R1 = R2 - value1
4:      .global label           //declaration of variable label
      SUB R1, R2, label         //R1 = R2 - label
      . . .
label:  nop                     //definition of variable label

```

AND - Logical AND of two operands**Syntax** AND *Rdst, Rsrc1, Rsrc2*AND *Rdst, Rsrc1, imm***Operands**

- *Rdst* - Register R[0..3]
- *Rsrc1* - Register R[0..3]
- *Rsrc2* - Register R[0..3]
- *Imm* - 16-bit signed value

Cycles 2 cycles to execute, 4 cycles to fetch next instruction**Description** The instruction does logical AND of a source register and another source register or 16-bit signed value and stores result to the destination register.**Examples:**

```

1:      AND R1, R2, R3          //R1 = R2 & R3
2:      AND R1, R2, 0x1234     //R1 = R2 & 0x1234
3:      .set value1, 0x03      //constant value1=0x03
      AND R1, R2, value1       //R1 = R2 & value1
4:      .global label          //declaration of variable label
      AND R1, R2, label        //R1 = R2 & label
      . . .
label:  nop                     //definition of variable label

```

OR - Logical OR of two operands**Syntax** OR *Rdst, Rsrc1, Rsrc2*OR *Rdst, Rsrc1, imm***Operands**

- *Rdst* - Register R[0..3]

- *Rsrc1* - Register R[0..3]
- *Rsrc2* - Register R[0..3]
- *Imm* - 16-bit signed value

Cycles 2 cycles to execute, 4 cycles to fetch next instruction

Description The instruction does logical OR of a source register and another source register or 16-bit signed value and stores result to the destination register.

Examples:

```

1:      OR R1, R2, R3           //R1 = R2 \| R3
2:      OR R1, R2, 0x1234      //R1 = R2 \| 0x1234
3:      .set value1, 0x03      //constant value1=0x03
      OR R1, R2, value1       //R1 = R2 \| value1
4:      .global label         //declaration of variable label
      OR R1, R2, label        //R1 = R2 \| label
      ...
label:  nop                   //definition of variable label

```

LSH - Logical Shift Left

Syntax **LSH** *Rdst, Rsrc1, Rsrc2*

LSH *Rdst, Rsrc1, imm*

Operands

- *Rdst* - Register R[0..3]
- *Rsrc1* - Register R[0..3]
- *Rsrc2* - Register R[0..3]
- *Imm* - 16-bit signed value

Cycles 2 cycles to execute, 4 cycles to fetch next instruction

Description The instruction does logical shift to left of source register to number of bits from another source register or 16-bit signed value and store result to the destination register.

Examples:

```

1:      LSH R1, R2, R3         //R1 = R2 << R3
2:      LSH R1, R2, 0x03      //R1 = R2 << 0x03
3:      .set value1, 0x03     //constant value1=0x03
      LSH R1, R2, value1     //R1 = R2 << value1
4:      .global label        //declaration of variable label
      LSH R1, R2, label      //R1 = R2 << label
      ...
label:  nop                   //definition of variable label

```

RSH - Logical Shift Right

Syntax **RSH** *Rdst, Rsrc1, Rsrc2*

RSH *Rdst, Rsrc1, imm*

Operands *Rdst* - Register R[0..3] *Rsrc1* - Register R[0..3] *Rsrc2* - Register R[0..3] *Imm* - 16-bit signed value

Cycles 2 cycles to execute, 4 cycles to fetch next instruction

Description The instruction does logical shift to right of source register to number of bits from another source register or 16-bit signed value and store result to the destination register.

Examples:

```

1:      RSH R1, R2, R3           //R1 = R2 >> R3
2:      RSH R1, R2, 0x03        //R1 = R2 >> 0x03
3:      .set value1, 0x03        //constant value1=0x03
      RSH R1, R2, value1        //R1 = R2 >> value1
4:      .global label           //declaration of variable label
      RSH R1, R2, label        //R1 = R2 >> label
label:  nop                     //definition of variable label

```

MOVE –Move to register

Syntax `MOVE Rdst, Rsrc`

`MOVE Rdst, imm`

Operands

- *Rdst* –Register R[0..3]
- *Rsrc* –Register R[0..3]
- *Imm* –16-bit signed value

Cycles 2 cycles to execute, 4 cycles to fetch next instruction

Description The instruction move to destination register value from source register or 16-bit signed value.

Note that when a label is used as an immediate, the address of the label will be converted from bytes to words.

This is because LD, ST, and JUMP instructions expect the address register value to be expressed in words rather than bytes. To avoid using an extra instruction

Examples:

```

1:      MOVE      R1, R2           //R1 = R2
2:      MOVE      R1, 0x03        //R1 = 0x03
3:      .set      value1, 0x03    //constant value1=0x03
      MOVE      R1, value1        //R1 = value1
4:      .global   label           //declaration of label
      MOVE      R1, label        //R1 = address_of(label) / 4
      ...
label:  nop                     //definition of label

```

STL/ST –Store data to the low 16 bits of 32-bits memory

Syntax `ST Rsrc, Rdst, offset, Label` `STL Rsrc, Rdst, offset, Label`

Operands

- *Rsrc* –Register R[0..3], holds the 16-bit value to store
- *Rdst* –Register R[0..3], address of the destination, in 32-bit words
- *Offset* –11-bit signed value, offset in bytes
- *Label* –2-bit user defined unsigned value

Cycles 4 cycles to execute, 4 cycles to fetch next instruction

Description The instruction stores the 16-bit value of Rsrc to the lower half-word of memory with address Rdst+offset:

```

Mem[Rdst + offset / 4]{15:0} = {Rsrc[15:0]}
Mem[Rdst + offset / 4]{15:0} = {Label[1:0], Rsrc[13:0]}

```

The ST command introduced to make compatibility with previous versions of UPL core. The application can use higher 16 bits to determine which instruction in the ULP program has written any particular word into memory.

Examples:

```

1:      STL   R1, R2, 0x12      //MEM[R2+0x12] = R1

2:      .data                      //Data section definition
Addr1:  .word   123              // Define label Addr1 16 bit
        .set    offs, 0x00      // Define constant offs
        .text                      //Text section definition
        MOVE   R1, 1            // R1 = 1
        MOVE   R2, Addr1        // R2 = Addr1
        STL    R1, R2, offs     // MEM[R2 + 0] = R1
                                      // MEM[Addr1 + 0] will be 32'hxxxx0001

3:      MOVE   R1, 1            // R1 = 1
        STL    R1, R2, 0x12,1   // MEM[R2+0x12] 0xxxxx4001

```

STH –Store data to the high 16 bits of 32-bits memory

Syntax `STH Rsrc, Rdst, offset, Label`

Operands

- *Rsrc* –Register R[0..3], holds the 16-bit value to store
- *Rdst* –Register R[0..3], address of the destination, in 32-bit words
- *Offset* –11-bit signed value, offset in bytes
- *Label* –2-bit user defined unsigned value

Cycles 4 cycles to execute, 4 cycles to fetch next instruction

Description The instruction stores the 16-bit value of *Rsrc* to the high half-word of memory with address *Rdst*+*offset*:

```

Mem[Rdst + offset / 4]{31:16} = {Rsrc[15:0]}
Mem[Rdst + offset / 4]{31:16} = {Label[1:0],Rsrc[13:0]}

```

Examples:

```

1:      STH   R1, R2, 0x12      //MEM[R2+0x12][31:16] = R1

2:      .data                      //Data section definition
Addr1:  .word   123              // Define label Addr1 16 bit
        .set    offs, 0x00      // Define constant offs
        .text                      //Text section definition
        MOVE   R1, 1            // R1 = 1
        MOVE   R2, Addr1        // R2 = Addr1
        STH    R1, R2, offs     // MEM[R2 + 0] = R1
                                      // MEM[Addr1 + 0] will be 32'h0001xxxx

3:      MOVE   R1, 1            // R1 = 1
        STH    R1, R2, 0x12, 1  //MEM[R2+0x12] 0x4001xxxx

```

STO –Set offset value for auto increment operation

Syntax `STO offset`

Operands

- *Offset* –11-bit signed value, offset in bytes

Cycles 4 cycles to execute, 4 cycles to fetch next instruction

Description The instruction set 16-bit value to the offset register:

```
offset = value / 4
```

Examples:

```

1:      STO   0x12           // Offset = 0x12/4

2:      .data                //Data section definition
Addr1:  .word   123          // Define label Addr1 16 bit
        .set    offs, 0x00   // Define constant offs
        .text                //Text section definition
        STO    offs         // Offset = 0x00

```

STI –Store data to the 32-bits memory with auto increment of predefined offset address

Syntax STI *Rsrc, Rdst, Label*

Operands

- *Rsrc* –Register R[0..3], holds the 16-bit value to store
- *Rdst* –Register R[0..3], address of the destination, in 32-bit words
- *Label* –2-bit user defined unsigned value

Cycles 4 cycles to execute, 4 cycles to fetch next instruction

Description The instruction stores the 16-bit value of *Rsrc* to the low and high half-word of memory with address *Rdst*+*offset* with auto increment of *offset*:

```

Mem[Rdst + offset / 4]{15:0/31:16} = {Rsrc[15:0]}
Mem[Rdst + offset / 4]{15:0/31:16} = {Label[1:0],Rsrc[13:0]}

```

Examples:

```

1:      STO   0              // Set offset to 0
        STI   R1, R2, 0x12  //MEM[R2+0x12][15:0] = R1
        STI   R1, R2, 0x12  //MEM[R2+0x12][31:16] = R1

2:      .data                //Data section definition
Addr1:  .word   123          // Define label Addr1 16 bit
        .set    offs, 0x00   // Define constant offs
        .text                //Text section definition
        STO    0             // Set offset to 0
        MOVE   R1, 1         // R1 = 1
        MOVE   R2, Addr1     // R2 = Addr1
        STI    R1, R2        // MEM[R2 + 0] = R1
                                // MEM[Addr1 + 0] will be 32'hxxxx0001
        STIx   R1, R2        // MEM[R2 + 0] = R1
                                // MEM[Addr1 + 0] will be 32'h00010001

3:      STO    0             // Set offset to 0
        MOVE   R1, 1         // R1 = 1
        STI    R1, R2, 1     //MEM[R2+0x12] 0xxxxx4001
        STI    R1, R2, 1     //MEM[R2+0x12] 0x40014001

```

ST32 –Store 32-bits data to the 32-bits memory

Syntax ST32 *Rsrc, Rdst, offset, Label*

Operands

- *Rsrc* –Register R[0..3], holds the 16-bit value to store
- *Rdst* –Register R[0..3], address of the destination, in 32-bit words
- *Offset* –11-bit signed value, offset in bytes
- *Label* –2-bit user defined unsigned value

Cycles 4 cycles to execute, 4 cycles to fetch next instruction

Description The instruction stores 11 bits of the PC value, label value and the 16-bit value of *Rsrc* to the 32-bits memory with address *Rdst*+*offset*:


```
Mem[Rdst + offset / 4]{31:0} = {PC[10:0],0[2:0],Label[1:0],Rsrc[15:0]}
```

Examples:

```
1:      STI32 R1, R2, 0x12, 0 //MEM[R2+0x12][31:0] = {PC[10:0],0[2:0],
↪Label[1:0],Rsrc[15:0]}

2:      .data //Data section definition
Addr1: .word 123 // Define label Addr1 16 bit
       .set offs, 0x00 // Define constant offs
       .text //Text section definition
       MOVE R1, 1 // R1 = 1
       MOVE R2, Addr1 // R2 = Addr1
       STI32 R1, R2, offs,1// MEM[R2 + 0] = {PC[10:0],0[2:0],Label[1:0],
↪Rsrc[15:0]}
// MEM[Addr1 + 0] will be 32'h00010001
```

STI32 –Store 32-bits data to the 32-bits memory with auto increment of adress offset**Syntax** STI32 Rsrc, Rdst, Label**Operands**

- *Rsrc* –Register R[0..3], holds the 16-bit value to store
- *Rdst* –Register R[0..3], address of the destination, in 32-bit words
- *Label* –2-bit user defined unsigned value

Cycles 4 cycles to execute, 4 cycles to fetch next instruction**Description** The instruction stores 11 bits of the PC value, label value and the 16-bit value of Rsrc to the 32-bits memory with address Rdst+offset:

```
Mem[Rdst + offset / 4]{31:0} = {PC[10:0],0[2:0],Label[1:0],Rsrc[15:0]}
```

Where offset value set by STO instruction

Examples:

```
1:      STO 0x12
       STI32 R1, R2, 0 //MEM[R2+0x12][31:0] = {PC[10:0],0[2:0],Label[1:0],
↪Rsrc[15:0]}
       STI32 R1, R2, 0 //MEM[R2+0x13][31:0] = {PC[10:0],0[2:0],Label[1:0],
↪Rsrc[15:0]}

2:      .data //Data section definition
Addr1: .word 123 // Define label Addr1 16 bit
       .set offs, 0x00 // Define constant offs
       .text //Text section definition
       MOVE R1, 1 // R1 = 1
       MOVE R2, Addr1 // R2 = Addr1
       STO offs
       STI32 R1, R2, 1// MEM[R2 + 0] = {PC[10:0],0[2:0],Label[1:0],
↪Rsrc[15:0]}
// MEM[Addr1 + 0] will be 32'h00010001
       STI32 R1, R2, 1// MEM[R2 + 1] = {PC[10:0],0[2:0],Label[1:0],
↪Rsrc[15:0]}
// MEM[Addr1 + 1] will be 32'h00010001
```

LDL/LD –Load data from low part of the 32-bits memory**Syntax** LD Rdst, Rsrc, offset LDL Rdst, Rsrc, offset**Operands** Rdst –Register R[0..3], destination*Rsrc* –Register R[0..3], holds address of destination, in 32-bit words*Offset* –10-bit signed value, offset in bytes

Cycles 4 cycles to execute, 4 cycles to fetch next instruction

Description The instruction loads lower 16-bit half-word from memory with address $Rsrc+offset$ into the destination register $Rdst$:

```
Rdst[15:0] = Mem[Rsrc + offset / 4][15:0]
```

The LD command do the same as LDL, and included for compatibility with previous versions of ULP core.

Examples:

```
1:      LDL  R1, R2, 0x12           //R1 = MEM[R2+0x12]
2:      .data                          //Data section definition
  Addr1: .word    123                 // Define label Addr1 16 bit
        .set     offs, 0x00          // Define constant offs
        .text                             //Text section definition
        MOVE    R1, 1                 // R1 = 1
        MOVE    R2, Addr1             // R2 = Addr1 / 4 (address of label is_
↳converted into words)
        LDL     R1, R2, offs          // R1 = MEM[R2 + 0]
                                           // R1 will be 123
```

LDH –Load data from high part of the 32-bits memory

Syntax LDH *Rdst, Rsrc, offset*

Operands *Rdst* –Register R[0..3], destination

Rsrc –Register R[0..3], holds address of destination, in 32-bit words

Offset –10-bit signed value, offset in bytes

Cycles 4 cycles to execute, 4 cycles to fetch next instruction

Description The instruction loads higher 16-bit half-word from memory with address $Rsrc+offset$ into the destination register $Rdst$:

```
Rdst[15:0] = Mem[Rsrc + offset / 4][15:0]
```

The LD command do the same as LDL, and included for compatibility with previous versions of ULP core.

Examples:

```
1:      LDH  R1, R2, 0x12           //R1 = MEM[R2+0x12]
2:      .data                          //Data section definition
  Addr1: .word    0x12345678          // Define label Addr1 16 bit
        .set     offs, 0x00          // Define constant offs
        .text                             //Text section definition
        MOVE    R1, 1                 // R1 = 1
        MOVE    R2, Addr1             // R2 = Addr1 / 4 (address of label is_
↳converted into words)
        LDH    R1, R2, offs          // R1 = MEM[R2 + 0]
                                           // R1 will be 0x1234
```

JUMP –Jump to an absolute address

Syntax JUMP *Rdst*

JUMP *ImmAddr*

JUMP *Rdst, Condition*

JUMP *ImmAddr, Condition*

Operands

- *Rdst* –Register R[0..3] containing address to jump to (expressed in 32-bit words)
- *ImmAddr* –13 bits address (expressed in bytes), aligned to 4 bytes
- **Condition:**

- EQ –jump if last ALU operation result was zero
- OV –jump if last ALU has set overflow flag

Cycles 2 cycles to execute, 2 cycles to fetch next instruction

Description The instruction makes jump to the specified address. Jump can be either unconditional or based on an ALU flag.

Examples:

```

1:      JUMP      R1          // Jump to address in R1 (address in R1 is in
↳32-bit words)

2:      JUMP      0x120, EQ   // Jump to address 0x120 (in bytes) if ALU
↳result is zero

3:      JUMP      label      // Jump to label
      ...
label:  nop                // Definition of label

4:      .global   label      // Declaration of global label

      MOVE      R1, label    // R1 = label (value loaded into R1 is in words)
      JUMP      R1          // Jump to label
      ...
label:  nop                // Definition of label

```

JUMPR –Jump to a relative offset (condition based on R0)

Syntax JUMPR *Step, Threshold, Condition*

Operands

- *Step* –relative shift from current position, in bytes
- *Threshold* –threshold value for branch condition
- **Condition:**
 - EQ (equal) –jump if value in R0 == threshold
 - LT (less than) –jump if value in R0 < threshold
 - LE (less or equal) –jump if value in R0 <= threshold
 - GT (greater than) –jump if value in R0 > threshold
 - GE (greater or equal) –jump if value in R0 >= threshold

Cycles Conditions EQ, GT and LT: 2 cycles to execute, 2 cycles to fetch next instruction

Conditions LE and GE are implemented in the assembler using two JUMPR instructions:

```

// JUMPR target, threshold, LE is implemented as:

      JUMPR target, threshold, EQ
      JUMPR target, threshold, LT

// JUMPR target, threshold, GE is implemented as:

      JUMPR target, threshold, EQ
      JUMPR target, threshold, GT

```

Therefore the execution time will depend on the branches taken: either 2 cycles to execute + 2 cycles to fetch, or 4 cycles to execute + 4 cycles to fetch.

Description The instruction makes a jump to a relative address if condition is true. Condition is the result of comparison of R0 register value and the threshold value.

Examples:

```

1:pos:   JUMPR      16, 20, GE // Jump to address (position + 16 bytes) if
↳value in R0 >= 20

2:      // Down counting loop using R0 register

```

(continues on next page)

```

label:  MOVE      R0, 16      // load 16 into R0
        SUB       R0, R0, 1   // R0--
        NOP                          // do something
        JUMPR    label, 1, GE // jump to label if R0 >= 1

```

JUMPS –Jump to a relative address (condition based on stage count)

Syntax JUMPS *Step, Threshold, Condition*

Operands

- *Step* –relative shift from current position, in bytes
- *Threshold* –threshold value for branch condition
- **Condition:**
 - *EQ* (equal) –jump if value in stage_cnt == threshold
 - *LT* (less than) –jump if value in stage_cnt < threshold
 - *LE* (less or equal) – jump if value in stage_cnt <= threshold
 - *GT* (greater than) –jump if value in stage_cnt > threshold
 - *GE* (greater or equal) —jump if value in stage_cnt >= threshold

Cycles 2 cycles to execute, 2 cycles to fetch next instruction:

```

// JUMPS target, threshold, EQ is implemented as:

        JUMPS next, threshold, LT
        JUMPS target, threshold, LE
next:

// JUMPS target, threshold, GT is implemented as:

        JUMPS next, threshold, LE
        JUMPS target, threshold, GE
next:

```

Therefore the execution time will depend on the branches taken: either 2 cycles to execute + 2 cycles to fetch, or 4 cycles to execute + 4 cycles to fetch.

Description The instruction makes a jump to a relative address if condition is true. Condition is the result of comparison of count register value and threshold value.

Examples:

```

1:pos:  JUMPS    16, 20, EQ    // Jump to (position + 16 bytes) if stage_cnt_
↪ == 20

2:      // Up counting loop using stage count register
        STAGE_RST                // set stage_cnt to 0
label:  STAGE_INC 1              // stage_cnt++
        NOP                      // do something
        JUMPS    label, 16, LT  // jump to label if stage_cnt < 16

```

STAGE_RST –Reset stage count register

Syntax STAGE_RST

Operands No operands

Description The instruction sets the stage count register to 0

Cycles 2 cycles to execute, 4 cycles to fetch next instruction

Examples:

```

1:      STAGE_RST                // Reset stage count register

```

STAGE_INC –Increment stage count register**Syntax** STAGE_INC *Value***Operands**

- *Value* –8 bits value

Cycles 2 cycles to execute, 4 cycles to fetch next instruction

Description The instruction increments stage count register by given value.

Examples:

```

1:      STAGE_INC      10          // stage_cnt += 10

2:      // Up counting loop example:
        STAGE_RST          // set stage_cnt to 0
label:  STAGE_INC      1          // stage_cnt++
        NOP                // do something
        JUMPS          label, 16, LT // jump to label if stage_cnt < 16

```

STAGE_DEC –Decrement stage count register**Syntax** STAGE_DEC *Value***Operands**

- *Value* –8 bits value

Cycles 2 cycles to execute, 4 cycles to fetch next instruction

Description The instruction decrements stage count register by given value.

Examples:

```

1:      STAGE_DEC      10          // stage_cnt -= 10;

2:      // Down counting loop exaple
        STAGE_RST          // set stage_cnt to 0
        STAGE_INC      16          // increment stage_cnt to 16
label:  STAGE_DEC      1          // stage_cnt--;
        NOP                // do something
        JUMPS          label, 0, GT // jump to label if stage_cnt > 0

```

HALT –End the program**Syntax** HALT

Operands No operands

Cycles 2 cycles to execute

Description The instruction halts the ULP coprocessor and restarts ULP wakeup timer, if it is enabled.

Examples:

```

1:      HALT          // Halt the coprocessor

```

WAKE –Wake up the chip**Syntax** WAKE

Operands No operands

Cycles 2 cycles to execute, 4 cycles to fetch next instruction

Description The instruction sends an interrupt from ULP to RTC controller.

- If the SoC is in deep sleep mode, and ULP wakeup is enabled, this causes the SoC to wake up.
- If the SoC is not in deep sleep mode, and ULP interrupt bit (RTC_CNTL_ULP_CP_INT_ENA) is set in RTC_CNTL_INT_ENA_REG register, RTC interrupt will be triggered.

Note that before using WAKE instruction, ULP program may needs to wait until RTC controller is ready to wake up the main CPU. This is indicated using RTC_CNTL_RDY_FOR_WAKEUP bit of RTC_CNTL_LOW_POWER_ST_REG register. If WAKE instruction is executed while RTC_CNTL_RDY_FOR_WAKEUP is zero, it has no effect (wake up does not occur).

Examples:

```

1: is_rdy_for_wakeup:                // Read RTC_CNTL_RDY_FOR_WAKEUP bit
    READ_RTC_FIELD(RTC_CNTL_LOW_POWER_ST_REG, RTC_CNTL_RDY_FOR_WAKEUP)
    AND r0, r0, 1
    JUMP is_rdy_for_wakeup, eq      // Retry until the bit is set
    WAKE                            // Trigger wake up
    REG_WR 0x006, 24, 24, 0        // Stop ULP timer (clear RTC_CNTL_ULP_CP_
↳SLP_TIMER_EN)
    HALT                            // Stop the ULP program
    // After these instructions, SoC will wake up,
    // and ULP will not run again until started by the main program.

```

SLEEP –set ULP wakeup timer period

Syntax SLEEP *sleep_reg*

Operands

- *sleep_reg* –0..4, selects one of SENS_ULP_CP_SLEEP_CYC_x_REG registers.

Cycles 2 cycles to execute, 4 cycles to fetch next instruction

Description The instruction selects which of the SENS_ULP_CP_SLEEP_CYC_x_REG (x = 0..4) register values is to be used by the ULP wakeup timer as wakeup period. By default, the value from SENS_ULP_CP_SLEEP_CYC0_REG is used.

Examples:

```

1:          SLEEP    1          // Use period set in SENS_ULP_CP_SLEEP_CYC1_REG

2:          .set sleep_reg, 4   // Set constant
           SLEEP    sleep_reg   // Use period set in SENS_ULP_CP_SLEEP_CYC4_REG

```

WAIT –wait some number of cycles

Syntax WAIT *Cycles*

Operands

- *Cycles* –number of cycles for wait

Cycles 2 + *Cycles* cycles to execute, 4 cycles to fetch next instruction

Description The instruction delays for given number of cycles.

Examples:

```

1:          WAIT     10         // Do nothing for 10 cycles

2:          .set  wait_cnt, 10  // Set a constant
           WAIT    wait_cnt     // wait for 10 cycles

```

TSENS –do measurement with temperature sensor

Syntax

- TSENS *Rdst*, *Wait_Delay*

Operands

- *Rdst* –Destination Register R[0..3], result will be stored to this register
- *Wait_Delay* –number of cycles used to perform the measurement

Cycles 2 + *Wait_Delay* + 3 * TSENS_CLK to execute, 4 cycles to fetch next instruction

Description The instruction performs measurement using TSENS and stores the result into a general purpose register.

Examples:

```
1:      TSENS      R1, 1000      // Measure temperature sensor for 1000 cycles,
                                     // and store result to R1
```

ADC –do measurement with ADC

Syntax

- **ADC** *Rdst, Sar_sel, Mux*
- **ADC** *Rdst, Sar_sel, Mux, 0* —deprecated form

Operands

- *Rdst* –Destination Register R[0..3], result will be stored to this register
- *Sar_sel* –Select ADC: 0 = SARADC1, 1 = SARADC2
- *Mux* - selected PAD, SARADC Pad[Mux+1] is enabled

Cycles $23 + \max(1, \text{SAR_AMP_WAIT1}) + \max(1, \text{SAR_AMP_WAIT2}) + \max(1, \text{SAR_AMP_WAIT3}) + \text{SARx_SAMPLE_CYCLE} + \text{SARx_SAMPLE_BIT}$ cycles to execute, 4 cycles to fetch next instruction

Description The instruction makes measurements from ADC.

Examples:

```
1:      ADC        R1, 0, 1      // Measure value using ADC1 pad 2 and store_
↳result into R1
```

I2C_RD - read single byte from I2C slave

Syntax

- **I2C_RD** *Sub_addr, High, Low, Slave_sel*

Operands

- *Sub_addr* –Address within the I2C slave to read.
- *High, Low* —Define range of bits to read. Bits outside of [High, Low] range are masked.
- *Slave_sel* - Index of I2C slave address to use.

Cycles Execution time mostly depends on I2C communication time. 4 cycles to fetch next instruction.

Description I2C_RD instruction reads one byte from I2C slave with index *Slave_sel*. Slave address (in 7-bit format) has to be set in advance into *SENS_I2C_SLAVE_ADDRx* register field, where $x == \text{Slave_sel}$. 8 bits of read result is stored into *R0* register.

Examples:

```
1:      I2C_RD     0x10, 7, 0, 0  // Read byte from sub-address 0x10 of_
↳slave with address set in SENS_I2C_SLAVE_ADDR0
```

I2C_WR - write single byte to I2C slave

Syntax

- **I2C_WR** *Sub_addr, Value, High, Low, Slave_sel*

Operands

- *Sub_addr* –Address within the I2C slave to write.
- *Value* –8-bit value to be written.
- *High, Low* —Define range of bits to write. Bits outside of [High, Low] range are masked.
- *Slave_sel* - Index of I2C slave address to use.

Cycles Execution time mostly depends on I2C communication time. 4 cycles to fetch next instruction.

Description I2C_WR instruction writes one byte to I2C slave with index *Slave_sel*. Slave address (in 7-bit format) has to be set in advance into *SENS_I2C_SLAVE_ADDRx* register field, where $x == \text{Slave_sel}$.

Examples:

```
1:      I2C_WR      0x20, 0x33, 7, 0, 1      // Write byte 0x33 to sub-address_
↳0x20 of slave with address set in SENS_I2C_SLAVE_ADDR1.
```

REG_RD –read from peripheral register**Syntax** REG_RD *Addr, High, Low***Operands**

- *Addr* –register address, in 32-bit words
- *High* –High part of R0
- *Low* –Low part of R0

Cycles 4 cycles to execute, 4 cycles to fetch next instruction**Description** The instruction reads up to 16 bits from a peripheral register into a general purpose register: $R0 = \text{REG}[\text{Addr}][\text{High}:\text{Low}]$.

This instruction can access registers in RTC_CNTL, RTC_IO, SENS, and RTC_I2C peripherals. Address of the the register, as seen from the ULP, can be calculated from the address of the same register on the DPORT bus as follows:

$$\text{addr_ulp} = (\text{addr_dport} - \text{DR_REG_RTCCNTL_BASE}) / 4$$
Examples:

```
1:      REG_RD      0x120, 2, 0      // load 4 bits: R0 = {12'b0, REG[0x120][7:4]}
```

REG_WR –write to peripheral register**Syntax** REG_WR *Addr, High, Low, Data***Operands**

- *Addr* –register address, in 32-bit words.
- *High* –High part of R0
- *Low* –Low part of R0
- *Data* –value to write, 8 bits

Cycles 8 cycles to execute, 4 cycles to fetch next instruction**Description** The instruction writes up to 8 bits from a general purpose register into a peripheral register. $\text{REG}[\text{Addr}][\text{High}:\text{Low}] = \text{data}$

This instruction can access registers in RTC_CNTL, RTC_IO, SENS, and RTC_I2C peripherals. Address of the the register, as seen from the ULP, can be calculated from the address of the same register on the DPORT bus as follows:

$$\text{addr_ulp} = (\text{addr_dport} - \text{DR_REG_RTCCNTL_BASE}) / 4$$
Examples:

```
1:      REG_WR      0x120, 7, 0, 0x10    // set 8 bits: REG[0x120][7:0] = 0x10
```

Convenience macros for peripheral registers access

ULP source files are passed through C preprocessor before the assembler. This allows certain macros to be used to facilitate access to peripheral registers.

Some existing macros are defined in `soc/soc_ulp.h` header file. These macros allow access to the fields of peripheral registers by their names. Peripheral registers names which can be used with these macros are the ones defined in `soc/rtc_cntl_reg.h`, `soc/rtc_io_reg.h`, `soc/sens_reg.h`, and `soc/rtc_i2c_reg.h`.

READ_RTC_REG(*rtc_reg*, *low_bit*, *bit_width*) Read up to 16 bits from *rtc_reg*[*low_bit* + *bit_width* - 1 : *low_bit*] into R0. For example:

```
#include "soc/soc_ulp.h"
#include "soc/rtc_cntl_reg.h"

/* Read 16 lower bits of RTC_CNTL_TIME0_REG into R0 */
READ_RTC_REG(RTC_CNTL_TIME0_REG, 0, 16)
```

READ_RTC_FIELD(*rtc_reg*, *field*) Read from a field in *rtc_reg* into R0, up to 16 bits. For example:

```
#include "soc/soc_ulp.h"
#include "soc/sens_reg.h"

/* Read 8-bit SENS_TSENS_OUT field of SENS_SAR_SLAVE_ADDR3_REG into R0 */
READ_RTC_FIELD(SENS_SAR_SLAVE_ADDR3_REG, SENS_TSENS_OUT)
```

WRITE_RTC_REG(*rtc_reg*, *low_bit*, *bit_width*, *value*) Write immediate value into *rtc_reg*[*low_bit* + *bit_width* - 1 : *low_bit*], *bit_width* <= 8. For example:

```
#include "soc/soc_ulp.h"
#include "soc/rtc_io_reg.h"

/* Set BIT(2) of RTC_GPIO_OUT_DATA_W1TS field in RTC_GPIO_OUT_W1TS_REG */
WRITE_RTC_REG(RTC_GPIO_OUT_W1TS_REG, RTC_GPIO_OUT_DATA_W1TS_S + 2, 1, 1)
```

WRITE_RTC_FIELD(*rtc_reg*, *field*, *value*) Write immediate value into a field in *rtc_reg*, up to 8 bits. For example:

```
#include "soc/soc_ulp.h"
#include "soc/rtc_cntl_reg.h"

/* Set RTC_CNTL_ULP_CP_SLP_TIMER_EN field of RTC_CNTL_STATE0_REG to 0 */
WRITE_RTC_FIELD(RTC_CNTL_STATE0_REG, RTC_CNTL_ULP_CP_SLP_TIMER_EN, 0)
```

5.29.3 Programming ULP coprocessor using C macros

In addition to the existing binutils port for the ESP32 ULP coprocessor, it is possible to generate programs for the ULP by embedding assembly-like macros into an ESP32 application. Here is an example how this can be done:

```
const ulp_insn_t program[] = {
    I_MOVI(R3, 16),          // R3 <- 16
    I_LD(R0, R3, 0),        // R0 <- RTC_SLOW_MEM[R3 + 0]
    I_LD(R1, R3, 1),        // R1 <- RTC_SLOW_MEM[R3 + 1]
    I_ADDR(R2, R0, R1),     // R2 <- R0 + R1
    I_ST(R2, R3, 2),        // R2 -> RTC_SLOW_MEM[R2 + 2]
    I_HALT()
};
size_t load_addr = 0;
size_t size = sizeof(program)/sizeof(ulp_insn_t);
ulp_process_macros_and_load(load_addr, program, &size);
ulp_run(load_addr);
```

The program array is an array of `ulp_insn_t`, i.e. ULP coprocessor instructions. Each `I_XXX` preprocessor define translates into a single 32-bit instruction. Arguments of these preprocessor defines can be register numbers (R0 —R3) and literal constants. See *ULP coprocessor instruction defines* section for descriptions of instructions and arguments they take.

Note: Because some of the instruction macros expand to inline function calls, defining such array in global scope will cause the compiler to produce an “initializer element is not constant” error. To fix this error, move the definition of instructions array into local scope.

Load and store instructions use addresses expressed in 32-bit words. Address 0 corresponds to the first word of RTC_SLOW_MEM (which is address 0x50000000 as seen by the main CPUs).

To generate branch instructions, special `M_` preprocessor defines are used. `M_LABEL` define can be used to define a branch target. Label identifier is a 16-bit integer. `M_Bxxx` defines can be used to generate branch instructions with target set to a particular label.

Implementation note: these `M_` preprocessor defines will be translated into two `ulp_insn_t` values: one is a token value which contains label number, and the other is the actual instruction. `ulp_process_macros_and_load` function resolves the label number to the address, modifies the branch instruction to use the correct address, and removes the the extra `ulp_insn_t` token which contains the label numer.

Here is an example of using labels and branches:

```
const ulp_insn_t program[] = {
    I_MOVI(R0, 34),          // R0 <- 34
    M_LABEL(1),             // label_1
    I_MOVI(R1, 32),          // R1 <- 32
    I_LD(R1, R1, 0),         // R1 <- RTC_SLOW_MEM[R1]
    I_MOVI(R2, 33),          // R2 <- 33
    I_LD(R2, R2, 0),         // R2 <- RTC_SLOW_MEM[R2]
    I_SUBR(R3, R1, R2),      // R3 <- R1 - R2
    I_ST(R3, R0, 0),         // R3 -> RTC_SLOW_MEM[R0 + 0]
    I_ADDI(R0, R0, 1),       // R0++
    M_BL(1, 64),            // if (R0 < 64) goto label_1
    I_HALT(),
};
RTC_SLOW_MEM[32] = 42;
RTC_SLOW_MEM[33] = 18;
size_t load_addr = 0;
size_t size = sizeof(program)/sizeof(ulp_insn_t);
ulp_process_macros_and_load(load_addr, program, &size);
ulp_run(load_addr);
```

Functions

esp_err_t `ulp_process_macros_and_load` (uint32_t *load_addr*, const ulp_insn_t **program*, size_t **psize*)

Resolve all macro references in a program and load it into RTC memory.

Return

- ESP_OK on success
- ESP_ERR_NO_MEM if auxiliary temporary structure can not be allocated
- one of ESP_ERR_ULP_xxx if program is not valid or can not be loaded

Parameters

- *load_addr*: address where the program should be loaded, expressed in 32-bit words
- *program*: `ulp_insn_t` array with the program
- *psize*: size of the program, expressed in 32-bit words

esp_err_t `ulp_run` (uint32_t *entry_point*)

Run the program loaded into RTC memory.

Return ESP_OK on success

Parameters

- *entry_point*: entry point, expressed in 32-bit words

Error codes

ESP_ERR_ULP_BASE

Offset for ULP-related error codes

ESP_ERR_ULP_SIZE_TOO_BIG

Program doesn't fit into RTC memory reserved for the ULP

ESP_ERR_ULP_INVALID_LOAD_ADDR

Load address is outside of RTC memory reserved for the ULP

ESP_ERR_ULP_DUPLICATE_LABEL

More than one label with the same number was defined

ESP_ERR_ULP_UNDEFINED_LABEL

Branch instructions references an undefined label

ESP_ERR_ULP_BRANCH_OUT_OF_RANGE

Branch target is out of range of B instruction (try replacing with BX)

ULP coprocessor registers

ULP co-processor has 4 16-bit general purpose registers. All registers have same functionality, with one exception. R0 register is used by some of the compare-and-branch instructions as a source register.

These definitions can be used for all instructions which require a register.

R0

general purpose register 0

R1

general purpose register 1

R2

general purpose register 2

R3

general purpose register 3

ULP coprocessor instruction defines

I_DELAY (cycles_)

Delay (nop) for a given number of cycles

I_HALT ()

Halt the coprocessor.

This instruction halts the coprocessor, but keeps ULP timer active. As such, ULP program will be restarted again by timer. To stop the program and prevent the timer from restarting the program, use I_END(0) instruction.

I_END ()

Stop ULP program timer.

This is a convenience macro which disables the ULP program timer. Once this instruction is used, ULP program will not be restarted anymore until ulp_run function is called.

ULP program will continue running after this instruction. To stop the currently running program, use I_HALT().

I_ST (reg_val, reg_addr, offset_)

Store value from register reg_val into RTC memory.

The value is written to an offset calculated by adding value of reg_addr register and offset_ field (this offset is expressed in 32-bit words). 32 bits written to RTC memory are built as follows:

- bits [31:21] hold the PC of current instruction, expressed in 32-bit words
- bits [20:16] = 5' b1
- bits [15:0] are assigned the contents of reg_val

RTC_SLOW_MEM[addr + offset_] = { 5' b0, insn_PC[10:0], val[15:0] }

I_LD (reg_dest, reg_addr, offset_)

Load value from RTC memory into reg_dest register.

Loads 16 LSBs from RTC memory word given by the sum of value in reg_addr and value of offset_.

I_WR_REG (reg, low_bit, high_bit, val)

Write literal value to a peripheral register

reg[high_bit : low_bit] = val This instruction can access RTC_CNTL_, RTC_IO_, SENS_, and RTC_I2C peripheral registers.

I_RD_REG (reg, low_bit, high_bit)

Read from peripheral register into R0

R0 = reg[high_bit : low_bit] This instruction can access RTC_CNTL_, RTC_IO_, SENS_, and RTC_I2C peripheral registers.

I_BL (pc_offset, imm_value)

Branch relative if R0 less than immediate value.

pc_offset is expressed in words, and can be from -127 to 127 imm_value is a 16-bit value to compare R0 against

I_BGE (pc_offset, imm_value)

Branch relative if R0 greater or equal than immediate value.

pc_offset is expressed in words, and can be from -127 to 127 imm_value is a 16-bit value to compare R0 against

I_BXR (reg_pc)

Unconditional branch to absolute PC, address in register.

reg_pc is the register which contains address to jump to. Address is expressed in 32-bit words.

I_BXI (imm_pc)

Unconditional branch to absolute PC, immediate address.

Address imm_pc is expressed in 32-bit words.

I_BXZR (reg_pc)

Branch to absolute PC if ALU result is zero, address in register.

reg_pc is the register which contains address to jump to. Address is expressed in 32-bit words.

I_BXZI (imm_pc)

Branch to absolute PC if ALU result is zero, immediate address.

Address imm_pc is expressed in 32-bit words.

I_BXFR (reg_pc)

Branch to absolute PC if ALU overflow, address in register

reg_pc is the register which contains address to jump to. Address is expressed in 32-bit words.

I_BXFI (imm_pc)

Branch to absolute PC if ALU overflow, immediate address

Address imm_pc is expressed in 32-bit words.

I_ADDR (reg_dest, reg_src1, reg_src2)

Addition: dest = src1 + src2

I_SUBR (reg_dest, reg_src1, reg_src2)

Subtraction: dest = src1 - src2

I_ANDR (reg_dest, reg_src1, reg_src2)

Logical AND: dest = src1 & src2

I_ORR (reg_dest, reg_src1, reg_src2)

Logical OR: dest = src1 | src2

I_MOVR (reg_dest, reg_src)

Copy: dest = src

I_LSHR (reg_dest, reg_src, reg_shift)

Logical shift left: dest = src << shift

I_RSHR (reg_dest, reg_src, reg_shift)

Logical shift right: dest = src >> shift

I_ADDI (reg_dest, reg_src, imm_)

Add register and an immediate value: dest = src + imm

I_SUBI (reg_dest, reg_src, imm_)

Subtract register and an immediate value: dest = src - imm

I_ANDI (reg_dest, reg_src, imm_)

Logical AND register and an immediate value: dest = src & imm

I_ORI (reg_dest, reg_src, imm_)

Logical OR register and an immediate value: dest = src | imm

I_MOVI (reg_dest, imm_)

Copy an immediate value into register: dest = imm

I_LSHI (reg_dest, reg_src, imm_)

Logical shift left register value by an immediate: dest = src << imm

I_RSHI (reg_dest, reg_src, imm_)

Logical shift right register value by an immediate: dest = val >> imm

M_LABEL (label_num)

Define a label with number label_num.

This is a macro which doesn't generate a real instruction. The token generated by this macro is removed by ulp_process_macros_and_load function. Label defined using this macro can be used in branch macros defined below.

M_BL (label_num, imm_value)

Macro: branch to label label_num if R0 is less than immediate value.

This macro generates two ulp_insn_t values separated by a comma, and should be used when defining contents of ulp_insn_t arrays. First value is not a real instruction; it is a token which is removed by ulp_process_macros_and_load function.

M_BGE (label_num, imm_value)

Macro: branch to label label_num if R0 is greater or equal than immediate value

This macro generates two ulp_insn_t values separated by a comma, and should be used when defining contents of ulp_insn_t arrays. First value is not a real instruction; it is a token which is removed by ulp_process_macros_and_load function.

M_BX (label_num)

Macro: unconditional branch to label

This macro generates two ulp_insn_t values separated by a comma, and should be used when defining contents of ulp_insn_t arrays. First value is not a real instruction; it is a token which is removed by ulp_process_macros_and_load function.

M_BXZ (label_num)

Macro: branch to label if ALU result is zero

This macro generates two ulp_insn_t values separated by a comma, and should be used when defining contents of ulp_insn_t arrays. First value is not a real instruction; it is a token which is removed by ulp_process_macros_and_load function.

M_BXF (label_num)

Macro: branch to label if ALU overflow

This macro generates two `ulp_insn_t` values separated by a comma, and should be used when defining contents of `ulp_insn_t` arrays. First value is not a real instruction; it is a token which is removed by `ulp_process_macros_and_load` function.

Defines

RTC_SLOW_MEM

RTC slow memory, 8k size

The ULP (Ultra Low Power) coprocessor is a simple FSM (Finite State Machine) which is designed to perform measurements using the ADC, temperature sensor, and external I2C sensors, while the main processors are in deep sleep mode. The ULP coprocessor can access the `RTC_SLOW_MEM` memory region, and registers in `RTC_CNTL`, `RTC_IO`, and `SARADC` peripherals. The ULP coprocessor uses fixed-width 32-bit instructions, 32-bit memory addressing, and has 4 general-purpose 16-bit registers.

5.29.4 Installing the Toolchain

The ULP coprocessor code is written in assembly and compiled using the `binutils-esp32ulp` toolchain.

If you have already set up ESP-IDF with CMake build system according to the [Getting Started Guide](#), then the ULP toolchain will already be installed.

If you are using ESP-IDF with the legacy GNU Make based build system, refer to the instructions on this page: [The ULP Coprocessor \(Legacy GNU Make\)](#).

5.29.5 Compiling the ULP Code

To compile the ULP code as part of the component, the following steps must be taken:

1. The ULP code, written in assembly, must be added to one or more files with `.S` extension. These files must be placed into a separate directory inside the component directory, for instance `ulp/`.
2. Call `ulp_embed_binary` from the component `CMakeLists.txt` after registration. For example:

```
...
idf_component_register()

set(ulp_app_name ulp_${COMPONENT_NAME})
set(ulp_s_sources ulp/ulp_assembly_source_file.S)
set(ulp_exp_dep_srcs "ulp_c_source_file.c")

ulp_embed_binary(${ulp_app_name} "${ulp_s_sources}" "${ulp_exp_dep_srcs}")
```

The first argument to `ulp_embed_binary` specifies the ULP binary name. The name specified here will also be used by other generated artifacts such as the ELF file, map file, header file and linker export file. The second argument specifies the ULP assembly source files. Finally, the third argument specifies the list of component source files which include the header file to be generated. This list is needed to build the dependencies correctly and ensure that the generated header file will be created before any of these files are compiled. See section below for the concept of generated header files for ULP applications.

3. Build the application as usual (e.g. `idf.py app`)
 Inside, the build system will take the following steps to build ULP program:
 1. **Run each assembly file (foo.S) through the C preprocessor.** This step generates the preprocessed assembly files (`foo.ulp.S`) in the component build directory. This step also generates dependency files (`foo.ulp.d`).
 2. **Run preprocessed assembly sources through the assembler.** This produces object (`foo.ulp.o`) and listing (`foo.ulp.lst`) files. Listing files are generated for debugging purposes and are not used at later stages of the build process.

3. **Run the linker script template through the C preprocessor.** The template is located in `components/ulp/ld` directory.
4. **Link the object files into an output ELF file** (`ulp_app_name.elf`). The Map file (`ulp_app_name.map`) generated at this stage may be useful for debugging purposes.
5. **Dump the contents of the ELF file into a binary** (`ulp_app_name.bin`) which can then be embedded into the application.
6. **Generate a list of global symbols** (`ulp_app_name.sym`) in the ELF file using `esp32ulp-elf-nm`.
7. **Create an LD export script and header file** (`ulp_app_name.ld` and `ulp_app_name.h`) containing the symbols from `ulp_app_name.sym`. This is done using the `esp32ulp_mapgen.py` utility.
8. **Add the generated binary to the list of binary files** to be embedded into the application.

5.29.6 Accessing the ULP Program Variables

Global symbols defined in the ULP program may be used inside the main program.

For example, the ULP program may define a variable `measurement_count` which will define the number of ADC measurements the program needs to make before waking up the chip from deep sleep:

```

                                .global measurement_count
measurement_count:              .long 0

                                /* later, use measurement_count */
                                move r3, measurement_count
                                ld r3, r3, 0

```

The main program needs to initialize this variable before the ULP program is started. The build system makes this possible by generating the `_${ULP_APP_NAME}.h` and `_${ULP_APP_NAME}.ld` files which define the global symbols present in the ULP program. Each global symbol defined in the ULP program is included in these files and are prefixed with `ulp_`.

The header file contains the declaration of the symbol:

```
extern uint32_t ulp_measurement_count;
```

Note that all symbols (variables, arrays, functions) are declared as `uint32_t`. For functions and arrays, take the address of the symbol and cast it to the appropriate type.

The generated linker script file defines the locations of symbols in `RTC_SLOW_MEM`:

```
PROVIDE ( ulp_measurement_count = 0x50000060 );
```

To access the ULP program variables from the main program, the generated header file should be included using an `include` statement. This will allow the ULP program variables to be accessed as regular variables:

```

#include "ulp_app_name.h"

// later
void init_ulp_vars() {
    ulp_measurement_count = 64;
}

```

Note that the ULP program can only use lower 16 bits of each 32-bit word in RTC memory, because the registers are 16-bit, and there is no instruction to load from the high part of the word.

Likewise, the ULP store instruction writes register value into the lower 16 bits part of the 32-bit word. The upper 16 bits are written with a value which depends on the address of the store instruction, thus when reading variables written by the ULP, the main application needs to mask the upper 16 bits, e.g.:

```
printf("Last measurement value: %d\n", ulp_last_measurement & UINT16_MAX);
```

5.29.7 Starting the ULP Program

To run a ULP program, the main application needs to load the ULP program into RTC memory using the `ulp_load_binary` function, and then start it using the `ulp_run` function.

Note that “Enable Ultra Low Power (ULP) Coprocessor” option must be enabled in menuconfig to reserve memory for the ULP. “RTC slow memory reserved for coprocessor” option must be set to a value sufficient to store ULP code and data. If the application components contain multiple ULP programs, then the size of the RTC memory must be sufficient to hold the largest one.

Each ULP program is embedded into the ESP-IDF application as a binary blob. The application can reference this blob and load it in the following way (suppose `ULP_APP_NAME` was defined to `ulp_app_name`):

```
extern const uint8_t bin_start[] asm("_binary_ulp_app_name_bin_start");
extern const uint8_t bin_end[]  asm("_binary_ulp_app_name_bin_end");

void start_ulp_program() {
    ESP_ERROR_CHECK( ulp_load_binary(
        0 /* load address, set to 0 when using default linker scripts */,
        bin_start,
        (bin_end - bin_start) / sizeof(uint32_t) );
}
```

esp_err_t **ulp_load_binary**(uint32_t *load_addr*, const uint8_t **program_binary*, size_t *program_size*)

Load ULP program binary into RTC memory.

ULP program binary should have the following format (all values little-endian):

1. MAGIC, (value 0x00706c75, 4 bytes)
2. TEXT_OFFSET, offset of .text section from binary start (2 bytes)
3. TEXT_SIZE, size of .text section (2 bytes)
4. DATA_SIZE, size of .data section (2 bytes)
5. BSS_SIZE, size of .bss section (2 bytes)
6. (TEXT_OFFSET - 12) bytes of arbitrary data (will not be loaded into RTC memory)
7. .text section
8. .data section

Linker script in components/ulp/ld/esp32.ulp.ld produces ELF files which correspond to this format. This linker script produces binaries with `load_addr == 0`.

Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_ARG` if `load_addr` is out of range
- `ESP_ERR_INVALID_SIZE` if `program_size` doesn't match (`TEXT_OFFSET + TEXT_SIZE + DATA_SIZE`)
- `ESP_ERR_NOT_SUPPORTED` if the magic number is incorrect

Parameters

- `load_addr`: address where the program should be loaded, expressed in 32-bit words
- `program_binary`: pointer to program binary
- `program_size`: size of the program binary

Once the program is loaded into RTC memory, the application can start it, passing the address of the entry point to the `ulp_run` function:

```
ESP_ERROR_CHECK( ulp_run(&ulp_entry - RTC_SLOW_MEM) );
```

esp_err_t **ulp_run**(uint32_t *entry_point*)

Run the program loaded into RTC memory.

Return ESP_OK on success

Parameters

- `entry_point`: entry point, expressed in 32-bit words

Declaration of the entry point symbol comes from the generated header file mentioned above, `#{ULP_APP_NAME}.h`. In the assembly source of the ULP application, this symbol must be marked as `.global`:

```
.global entry
entry:
    /* code starts here */
```

5.29.8 ULP Program Flow

The ULP coprocessor is started by a timer. The timer is started once `ulp_run` is called. The timer counts a number of `RTC_SLOW_CLK` ticks (by default, produced by an internal 150 kHz RC oscillator). The number of ticks is set using `SENS_ULP_CP_SLEEP_CYCx_REG` registers ($x = 0..4$). When starting the ULP for the first time, `SENS_ULP_CP_SLEEP_CYC0_REG` will be used to set the number of timer ticks. Later the ULP program can select another `SENS_ULP_CP_SLEEP_CYCx_REG` register using the `sleep` instruction.

The application can set ULP timer period values (`SENS_ULP_CP_SLEEP_CYCx_REG`, $x = 0..4$) using the `ulp_set_wakeup_period` function.

`esp_err_t ulp_set_wakeup_period` (`size_t period_index`, `uint32_t period_us`)

Set one of ULP wakeup period values.

ULP coprocessor starts running the program when the wakeup timer counts up to a given value (called period). There are 5 period values which can be programmed into `SENS_ULP_CP_SLEEP_CYCx_REG` registers, $x = 0..4$. By default, wakeup timer will use the period set into `SENS_ULP_CP_SLEEP_CYC0_REG`, i.e. period number 0. ULP program code can use `SLEEP` instruction to select which of the `SENS_ULP_CP_SLEEP_CYCx_REG` should be used for subsequent wakeups.

However, please note that `SLEEP` instruction issued (from ULP program) while the system is in deep sleep mode does not have effect, and sleep cycle count 0 is used.

Note The ULP FSM requires two clock cycles to wakeup before being able to run the program. Then additional 16 cycles are reserved after wakeup waiting until the 8M clock is stable. The FSM also requires two more clock cycles to go to sleep after the program execution is halted. The minimum wakeup period that may be set up for the ULP is equal to the total number of cycles spent on the above internal tasks. For a default configuration of the ULP running at 150kHz it makes about 133us.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if `period_index` is out of range

Parameters

- `period_index`: wakeup period setting number (0 - 4)
- `period_us`: wakeup period, us

Once the timer counts the number of ticks set in the selected `SENS_ULP_CP_SLEEP_CYCx_REG` register, the ULP coprocessor will power up and start running the program from the entry point set in the call to `ulp_run`.

The program runs until it encounters a `halt` instruction or an illegal instruction. Once the program halts, the ULP coprocessor will power down, and the timer will be started again.

To disable the timer (effectively preventing the ULP program from running again), please clear the `RTC_CNTL_ULP_CP_SLP_TIMER_EN` bit in the `RTC_CNTL_STATE0_REG` register. This can be done both from the ULP code and from the main program.

5.30 The ULP Coprocessor (Legacy GNU Make)

Note: Since ESP-IDF V4.0, the default build system is based on CMake. This documentation is for the legacy build system based on GNU Make. Support for this build system may be removed in future major releases.

The ULP (Ultra Low Power) coprocessor is a simple FSM (Finite State Machine) which is designed to perform measurements using the ADC, temperature sensor, and external I2C sensors, while the main processors are in deep sleep mode. The ULP coprocessor can access the RTC_SLOW_MEM memory region, and registers in RTC_CNTL, RTC_IO, and SARADC peripherals. The ULP coprocessor uses fixed-width 32-bit instructions, 32-bit memory addressing, and has 4 general-purpose 16-bit registers.

5.30.1 Installing the Toolchain

The ULP coprocessor code is written in assembly and compiled using the *binutils-esp32ulp toolchain*.

1. Download pre-built binaries of the latest toolchain release from: <https://github.com/espressif/binutils-esp32ulp/releases>.
2. Extract the toolchain into a directory, and add the path to the `bin/` directory of the toolchain to the `PATH` environment variable.

5.30.2 Compiling the ULP Code

To compile the ULP code as part of the component, the following steps must be taken:

1. The ULP code, written in assembly, must be added to one or more files with `.S` extension. These files must be placed into a separate directory inside the component directory, for instance *ulp/*.
2. Modify the component makefile, adding the following:

```

ULP_APP_NAME ?= ulp_$(COMPONENT_NAME)
ULP_S_SOURCES = $(COMPONENT_PATH)/ulp/ulp_source_file.S
ULP_EXP_DEP_OBJECTS := main.o
include $(IDF_PATH)/components/ulp/component_ulp_common.mk

```

Here is each line explained:

ULP_APP_NAME Name of the generated ULP application, without an extension. This name is used for build products of the ULP application: ELF file, map file, binary file, generated header file, and generated linker export file.

ULP_S_SOURCES List of assembly files to be passed to the ULP assembler. These must be absolute paths, i.e. start with `$(COMPONENT_PATH)`. Consider using `$(addprefix)` function if more than one file needs to be listed. Paths are relative to component build directory, so prefixing them is not necessary.

ULP_EXP_DEP_OBJECTS List of object files names within the component which include the generated header file. This list is needed to build the dependencies correctly and ensure that the generated header file is created before any of these files are compiled. See section below explaining the concept of generated header files for ULP applications.

include `$(IDF_PATH)/components/ulp/component_ulp_common.mk` Includes common definitions of ULP build steps. Defines build targets for ULP object files, ELF file, binary file, etc.

3. Build the application as usual (e.g. `idf.py build` or `idf.py app`)

Inside, the build system will take the following steps to build ULP program:

1. **Run each assembly file (`foo.S`) through the C preprocessor.** This step generates the preprocessed assembly files (`foo.ulp.ps`) in the component build directory. This step also generates dependency files (`foo.ulp.d`).
2. **Run the preprocessed assembly sources through the assembler.** This produces object (`foo.ulp.o`) and listing (`foo.ulp.lst`) files. Listing files are generated for debugging purposes and are not used at later stages of the build process.

3. **Run the linker script template through the C preprocessor.** The template is located in `components/ulp/ld` directory.
4. **Link the object files into an output ELF file** (`ulp_app_name.elf`). The Map file (`ulp_app_name.map`) generated at this stage may be useful for debugging purposes.
5. **Dump the contents of the ELF file into a binary** (`ulp_app_name.bin`) which can then be embedded into the application.
6. **Generate a list of global symbols** (`ulp_app_name.sym`) in the ELF file using `esp32ulp-elf-nm`.
7. **Create an LD export script and header file** (`ulp_app_name.ld` and `ulp_app_name.h`) containing the symbols from `ulp_app_name.sym`. This is done using the `esp32ulp_mapgen.py` utility.
8. **Add the generated binary to the list of binary files** to be embedded into the application.

5.30.3 Accessing the ULP Program Variables

Global symbols defined in the ULP program may be used inside the main program.

For example, the ULP program may define a variable `measurement_count` which will define the number of the ADC measurements the program needs to make before waking up the chip from deep sleep:

```

                                .global measurement_count
measurement_count:              .long 0

                                /* later, use measurement_count */
                                move r3, measurement_count
                                ld r3, r3, 0

```

The main program needs to initialize this variable before the ULP program is started. The build system makes this possible by generating `$(ULP_APP_NAME).h` and `$(ULP_APP_NAME).ld` files which define global symbols present in the ULP program. Each global symbol defined in the ULP program is included in these files and are prefixed with `ulp_`.

The header file contains the declaration of the symbol:

```
extern uint32_t ulp_measurement_count;
```

Note that all symbols (variables, arrays, functions) are declared as `uint32_t`. For functions and arrays, take the address of the symbol and cast it to the appropriate type.

The generated linker script file defines locations of symbols in `RTC_SLOW_MEM`:

```
PROVIDE ( ulp_measurement_count = 0x50000060 );
```

To access the ULP program variables from the main program, the generated header file should be included using an `include` statement. This will allow the ULP program variables to be accessed as regular variables:

```

#include "ulp_app_name.h"

// later
void init_ulp_vars() {
    ulp_measurement_count = 64;
}

```

Note that the ULP program can only use lower 16 bits of each 32-bit word in RTC memory, because the registers are 16-bit, and there is no instruction to load from the high part of the word.

Likewise, the ULP store instruction writes register value into the lower 16 bits part of the 32-bit word. The upper 16 bits are written with a value which depends on the address of the store instruction, thus when reading variables written by the ULP, the main application needs to mask the upper 16 bits, e.g.:

```
printf("Last measurement value: %d\n", ulp_last_measurement & UINT16_MAX);
```

5.30.4 Starting the ULP Program

To run a ULP program, the main application needs to load the ULP program into RTC memory using the `ulp_load_binary` function, and then start it using the `ulp_run` function.

Note that “Enable Ultra Low Power (ULP) Coprocessor” option must be enabled in menuconfig to reserve memory for the ULP. “RTC slow memory reserved for coprocessor” option must be set to a value sufficient to store ULP code and data. If the application components contain multiple ULP programs, then the size of the RTC memory must be sufficient to hold the largest one.

Each ULP program is embedded into the ESP-IDF application as a binary blob. The application can reference this blob and load it in the following way (suppose `ULP_APP_NAME` was defined to `ulp_app_name`):

```
extern const uint8_t bin_start[] asm("_binary_ulp_app_name_bin_start");
extern const uint8_t bin_end[]   asm("_binary_ulp_app_name_bin_end");

void start_ulp_program() {
    ESP_ERROR_CHECK( ulp_load_binary(
        0 /* load address, set to 0 when using default linker scripts */,
        bin_start,
        (bin_end - bin_start) / sizeof(uint32_t) );
}
```

esp_err_t **ulp_load_binary**(uint32_t *load_addr*, const uint8_t **program_binary*, size_t *program_size*)

Load ULP program binary into RTC memory.

ULP program binary should have the following format (all values little-endian):

1. MAGIC, (value 0x00706c75, 4 bytes)
2. TEXT_OFFSET, offset of .text section from binary start (2 bytes)
3. TEXT_SIZE, size of .text section (2 bytes)
4. DATA_SIZE, size of .data section (2 bytes)
5. BSS_SIZE, size of .bss section (2 bytes)
6. (TEXT_OFFSET - 12) bytes of arbitrary data (will not be loaded into RTC memory)
7. .text section
8. .data section

Linker script in components/ulp/ld/esp32.ulp.ld produces ELF files which correspond to this format. This linker script produces binaries with `load_addr == 0`.

Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_ARG` if `load_addr` is out of range
- `ESP_ERR_INVALID_SIZE` if `program_size` doesn't match (`TEXT_OFFSET + TEXT_SIZE + DATA_SIZE`)
- `ESP_ERR_NOT_SUPPORTED` if the magic number is incorrect

Parameters

- `load_addr`: address where the program should be loaded, expressed in 32-bit words
- `program_binary`: pointer to program binary
- `program_size`: size of the program binary

Once the program is loaded into RTC memory, the application can start it, passing the address of the entry point to the `ulp_run` function:

```
ESP_ERROR_CHECK( ulp_run(&ulp_entry - RTC_SLOW_MEM) );
```

esp_err_t **ulp_run**(uint32_t *entry_point*)

Run the program loaded into RTC memory.

Return ESP_OK on success

Parameters

- `entry_point`: entry point, expressed in 32-bit words

Declaration of the entry point symbol comes from the generated header file mentioned above, `$(ULP_APP_NAME).h`. In the assembly source of the ULP application, this symbol must be marked as `.global`:

```
.global entry
entry:
    /* code starts here */
```

5.30.5 ULP Program Flow

The ULP coprocessor is started by a timer. The timer is started once `ulp_run` is called. The timer counts the number of `RTC_SLOW_CLK` ticks (by default, produced by an internal 150 kHz RC oscillator). The number of ticks is set using `SENS_ULP_CP_SLEEP_CYCx_REG` registers ($x = 0..4$). When starting the ULP for the first time, `SENS_ULP_CP_SLEEP_CYC0_REG` will be used to set the number of timer ticks. Later the ULP program can select another `SENS_ULP_CP_SLEEP_CYCx_REG` register using the `sleep` instruction.

The application can set ULP timer period values (`SENS_ULP_CP_SLEEP_CYCx_REG`, $x = 0..4$) using the `ulp_set_wakeup_period` function.

`esp_err_t ulp_set_wakeup_period(size_t period_index, uint32_t period_us)`

Set one of ULP wakeup period values.

ULP coprocessor starts running the program when the wakeup timer counts up to a given value (called period). There are 5 period values which can be programmed into `SENS_ULP_CP_SLEEP_CYCx_REG` registers, $x = 0..4$. By default, wakeup timer will use the period set into `SENS_ULP_CP_SLEEP_CYC0_REG`, i.e. period number 0. ULP program code can use `SLEEP` instruction to select which of the `SENS_ULP_CP_SLEEP_CYCx_REG` should be used for subsequent wakeups.

However, please note that `SLEEP` instruction issued (from ULP program) while the system is in deep sleep mode does not have effect, and sleep cycle count 0 is used.

Note The ULP FSM requires two clock cycles to wakeup before being able to run the program. Then additional 16 cycles are reserved after wakeup waiting until the 8M clock is stable. The FSM also requires two more clock cycles to go to sleep after the program execution is halted. The minimum wakeup period that may be set up for the ULP is equal to the total number of cycles spent on the above internal tasks. For a default configuration of the ULP running at 150kHz it makes about 133us.

Return

- ESP_OK on success
- ESP_ERR_INVALID_ARG if `period_index` is out of range

Parameters

- `period_index`: wakeup period setting number (0 - 4)
- `period_us`: wakeup period, us

Once the timer counts the number of ticks set in the selected `SENS_ULP_CP_SLEEP_CYCx_REG` register, the ULP coprocessor will power up and start running the program from the entry point set in the call to `ulp_run`.

The program runs until it encounters a `halt` instruction or an illegal instruction. Once the program halts, ULP coprocessor will power down, and the timer will be started again.

To disable the timer (effectively preventing the ULP program from running again), please clear the `RTC_CNTL_ULP_CP_SLP_TIMER_EN` bit in the `RTC_CNTL_STATE0_REG` register. This can be done both from the ULP code and from the main program.

5.31 Unit Testing (Legacy GNU Make)

Note: Since ESP-IDF V4.0, the default build system is based on CMake. This documentation is for the legacy build system based on GNU Make. Support for this build system may be removed in future major releases.

ESP-IDF comes with a unit test application that is based on the Unity - unit test framework. Unit tests are integrated in the ESP-IDF repository and are placed in the `test` subdirectories of each component respectively.

5.31.1 Normal Test Cases

Unit tests are located in the `test` subdirectory of a component. Tests are added in C files, a single C file can include multiple test cases. Test files start with the word “test” .

Each test file should include the `unity.h` header and the header for the C module to be tested.

Tests are added in a function in the C file as follows:

```
TEST_CASE("test name", "[module name]"
{
    // Add test here
})
```

The first argument is a descriptive name for the test, the second argument is an identifier in square brackets. Identifiers are used to group related test, or tests with specific properties.

Note: There is no need to add a main function with `UNITY_BEGIN()` and `UNITY_END()` in each test case. `unity_platform.c` will run `UNITY_BEGIN()` autonomously, and run the test cases, then call `UNITY_END()`.

Each `test` subdirectory needs to include a `component.mk` file with the following line of code:

```
COMPONENT_ADD_LDFLAGS = -Wl,--whole-archive -l$(COMPONENT_NAME) -Wl,--no-whole-
↪archive
```

See <http://www.throwtheswitch.org/unity> for more information about writing tests in Unity.

5.31.2 Multi-device Test Cases

The normal test cases will be executed on one DUT (Device Under Test). However, components that require some form of communication (e.g., GPIO, SPI) require another device to communicate with, thus cannot be tested normal test cases. Multi-device test cases involve writing multiple test functions, and running them on multiple DUTs.

The following is an example of a Multi-device test case:

```
void gpio_master_test()
{
    gpio_config_t slave_config = {
        .pin_bit_mask = 1 << MASTER_GPIO_PIN,
        .mode = GPIO_MODE_INPUT,
    };
    gpio_config(&slave_config);
    unity_wait_for_signal("output high level");
    TEST_ASSERT(gpio_get_level(MASTER_GPIO_PIN) == 1);
}

void gpio_slave_test()
```

(continues on next page)

(continued from previous page)

```

{
    gpio_config_t master_config = {
        .pin_bit_mask = 1 << SLAVE_GPIO_PIN,
        .mode = GPIO_MODE_OUTPUT,
    };
    gpio_config(&master_config);
    gpio_set_level(SLAVE_GPIO_PIN, 1);
    unity_send_signal("output high level");
}

TEST_CASE_MULTIPLE_DEVICES("gpio multiple devices test example", "[driver]", gpio_
↪master_test, gpio_slave_test);

```

The macro `TEST_CASE_MULTIPLE_DEVICES` is used to declare a multi-device test case. The first argument is test case name, the second argument is test case description. From the third argument, up to 5 test functions can be defined, each function will be the entry point of tests running on each DUT.

Running test cases from different DUTs could require synchronizing between DUTs. We provide `unity_wait_for_signal` and `unity_send_signal` to support synchronizing with UART. As the scenario in the above example, the slave should get GPIO level after master set level. DUT UART console will prompt and requires user interaction:

DUT1 (master) console:

```

Waiting for signal: [output high level]!
Please press "Enter" key once any board send this signal.

```

DUT2 (slave) console:

```

Send signal: [output high level]!

```

Once the signal is sent from DUT2, you need to press “Enter” on DUT1, then DUT1 unblocks from `unity_wait_for_signal` and starts to change GPIO level.

Signals can also be used to pass parameters between multiple devices. For example, DUT1 want to know the MAC address of DUT2, so it can connect to DUT2. In this case, `unity_wait_for_signal_param` and `unity_send_signal_param` can be used:

DUT1 console:

```

Waiting for signal: [dut2 mac address]!
Please input parameter value from any board send this signal and press "Enter" key.

```

DUT2 console:

```

Send signal: [dut2 mac address][10:20:30:40:50:60]!

```

Once the signal is sent from DUT2, you need to input `10:20:30:40:50:60` on DUT1 and press “Enter”. Then DUT1 will get the MAC address string of DUT2 and unblock from `unity_wait_for_signal_param`, then start to connect to DUT2.

5.31.3 Multi-stage Test Cases

The normal test cases are expected to finish without reset (or only need to check if reset happens). Sometimes we expect to run some specific tests after certain kinds of reset. For example, we expect to test if reset reason is correct after a wakeup from deep sleep. We need to create a deep-sleep reset first and then check the reset reason. To support this, we can define multi-stage test cases, to group a set of test functions:


```

static void trigger_deepsleep(void)
{
    esp_sleep_enable_timer_wakeup(2000);
    esp_deep_sleep_start();
}

void check_deepsleep_reset_reason()
{
    RESET_REASON reason = rtc_get_reset_reason(0);
    TEST_ASSERT(reason == DEEPSLEEP_RESET);
}

TEST_CASE_MULTIPLE_STAGES("reset reason check for deepsleep", "[esp32]", trigger_
↪deepsleep, check_deepsleep_reset_reason);

```

Multi-stage test cases present a group of test functions to users. It need user interactions (select cases and select different stages) to run the case.

5.31.4 Building Unit Test App

Follow the setup instructions in the top-level esp-idf README. Make sure that IDF_PATH environment variable is set to point to the path of esp-idf top-level directory.

Change into `tools/unit-test-app` directory to configure and build it:

- `make menuconfig` - configure unit test app.
- `make TESTS_ALL=1` - build unit test app with tests for each component having tests in the `test` subdirectory.
- `make TEST_COMPONENTS='xxx'` - build unit test app with tests for specific components.
- `make TESTS_ALL=1 TEST_EXCLUDE_COMPONENTS='xxx'` - build unit test app with all unit tests, except for unit tests of some components. (For instance: `make TESTS_ALL=1 TEST_EXCLUDE_COMPONENTS='ulp mbedtls'` - build all unit tests excludes `ulp` and `mbedtls` components).

When the build finishes, it will print instructions for flashing the chip. You can simply run `make flash` to flash all build output.

You can also run `make flash TESTS_ALL=1` or `make TEST_COMPONENTS='xxx'` to build and flash. Everything needed will be rebuilt automatically before flashing.

Use `menuconfig` to set the serial port for flashing.

5.31.5 Running Unit Tests

After flashing reset the ESP32 and it will boot the unit test app.

When unit test app is idle, press “Enter” will make it print test menu with all available tests:

```

Here's the test menu, pick your combo:
(1)  "esp_ota_begin() verifies arguments" [ota]
(2)  "esp_ota_get_next_update_partition logic" [ota]
(3)  "Verify bootloader image in flash" [bootloader_support]
(4)  "Verify unit test app image" [bootloader_support]
(5)  "can use new and delete" [cxx]
(6)  "can call virtual functions" [cxx]
(7)  "can use static initializers for non-POD types" [cxx]
(8)  "can use std::vector" [cxx]
(9)  "static initialization guards work as expected" [cxx]
(10) "global initializers run in the correct order" [cxx]
(11) "before scheduler has started, static initializers work correctly" [cxx]

```

(continues on next page)

(continued from previous page)

```

(12)  "adc2 work with wifi" [adc]
(13)  "gpio master/slave test example" [ignore][misc][test_env=UT_T2_1][multi_
↪device]
      (1)    "gpio_master_test"
      (2)    "gpio_slave_test"
(14)  "SPI Master clockdiv calculation routines" [spi]
(15)  "SPI Master test" [spi][ignore]
(16)  "SPI Master test, interaction of multiple devs" [spi][ignore]
(17)  "SPI Master no response when switch from host1 (HSPI) to host2 (VSPI)" ↪
↪[spi]
(18)  "SPI Master DMA test, TX and RX in different regions" [spi]
(19)  "SPI Master DMA test: length, start, not aligned" [spi]
(20)  "reset reason check for deepsleep" [esp32][test_env=UT_T2_1][multi_stage]
      (1)    "trigger_deepsleep"
      (2)    "check_deepsleep_reset_reason"

```

The normal case will print the case name and description. Master-slave cases will also print the sub-menu (the registered test function names).

Test cases can be run by inputting one of the following:

- Test case name in quotation marks (for example, "esp_ota_begin() verifies arguments") to run a single test case.
- Test case index (for example, 1) to run a single test case.
- Module name in square brackets (for example, [cxx]) to run all test cases for a specific module.
- An asterisk (*) to run all test cases

[multi_device] and [multi_stage] tags tell the test runner whether a test case is a multi-device or multi-stage test case. These tags are automatically added by `TEST_CASE_MULTIPLE_STAGES` and `TEST_CASE_MULTIPLE_DEVICES` macros.

After you select a multi-device test case, it will print sub menu:

```

Running gpio master/slave test example...
gpio master/slave test example
      (1)    "gpio_master_test"
      (2)    "gpio_slave_test"

```

You need to input a number to select the test running on the DUT.

Similar to multi-device test cases, multi-stage test cases will also print sub-menu:

```

Running reset reason check for deepsleep...
reset reason check for deepsleep
      (1)    "trigger_deepsleep"
      (2)    "check_deepsleep_reset_reason"

```

For the first time you execute this case, please input 1 to run the first stage (trigger deep-sleep). After DUT is rebooted and test cases are available to run, select this case again and input 2 to run the second stage. The case will only pass if the last stage passes and all previous stages trigger reset.

5.32 Unit Testing in ESP32

ESP-IDF comes with a unit test application that is based on the Unity - unit test framework. Unit tests are integrated in the ESP-IDF repository and are placed in the `test` subdirectories of each component respectively.

5.32.1 Normal Test Cases

Unit tests are located in the `test` subdirectory of a component. Tests are written in C, and a single C source file can contain multiple test cases. Test files start with the word “test” .

Each test file should include the `unity.h` header and the header for the C module to be tested.

Tests are added in a function in the C file as follows:

```
TEST_CASE("test name", "[module name]"
{
    // Add test here
})
```

- The first argument is a descriptive name for the test.
- The second argument is an identifier in square brackets. Identifiers are used to group related test, or tests with specific properties.

Note: There is no need to add a main function with `UNITY_BEGIN()` and `UNITY_END()` in each test case. `unity_platform.c` will run `UNITY_BEGIN()` autonomously, and run the test cases, then call `UNITY_END()`.

The `test` subdirectory should contain a *component CMakeLists.txt*, since they are themselves, components. ESP-IDF uses the `unity` test framework and should be specified as a requirement for the component. Normally, components *should list their sources manually*; for component tests however, this requirement is relaxed and the use of the `SRC_DIRS` argument in `idf_component_register` is advised.

Overall, the minimal `test` subdirectory `CMakeLists.txt` file should contain the following:

```
idf_component_register(SRC_DIRS "."
                      INCLUDE_DIRS "."
                      REQUIRES unity)
```

See <http://www.throwtheswitch.org/unity> for more information about writing tests in Unity.

5.32.2 Multi-device Test Cases

The normal test cases will be executed on one DUT (Device Under Test). However, components that require some form of communication (e.g., GPIO, SPI) require another device to communicate with, thus cannot be tested normal test cases. Multi-device test cases involve writing multiple test functions, and running them on multiple DUTs.

The following is an example of a multi-device test case:

```
void gpio_master_test()
{
    gpio_config_t slave_config = {
        .pin_bit_mask = 1 << MASTER_GPIO_PIN,
        .mode = GPIO_MODE_INPUT,
    };
    gpio_config(&slave_config);
    unity_wait_for_signal("output high level");
    TEST_ASSERT(gpio_get_level(MASTER_GPIO_PIN) == 1);
}

void gpio_slave_test()
{
    gpio_config_t master_config = {
        .pin_bit_mask = 1 << SLAVE_GPIO_PIN,
        .mode = GPIO_MODE_OUTPUT,
    };
};
```

(continues on next page)

(continued from previous page)

```

gpio_config(&master_config);
gpio_set_level(SLAVE_GPIO_PIN, 1);
unity_send_signal("output high level");
}

TEST_CASE_MULTIPLE_DEVICES("gpio multiple devices test example", "[driver]", gpio_
↪master_test, gpio_slave_test);

```

The macro `TEST_CASE_MULTIPLE_DEVICES` is used to declare a multi-device test case.

- The first argument is test case name.
- The second argument is test case description.
- From the third argument, up to 5 test functions can be defined, each function will be the entry point of tests running on each DUT.

Running test cases from different DUTs could require synchronizing between DUTs. We provide `unity_wait_for_signal` and `unity_send_signal` to support synchronizing with UART. As the scenario in the above example, the slave should get GPIO level after master set level. DUT UART console will prompt and user interaction is required:

DUT1 (master) console:

```

Waiting for signal: [output high level]!
Please press "Enter" key to once any board send this signal.

```

DUT2 (slave) console:

```

Send signal: [output high level]!

```

Once the signal is sent from DUT2, you need to press “Enter” on DUT1, then DUT1 unblocks from `unity_wait_for_signal` and starts to change GPIO level.

5.32.3 Multi-stage Test Cases

The normal test cases are expected to finish without reset (or only need to check if reset happens). Sometimes we expect to run some specific tests after certain kinds of reset. For example, we expect to test if the reset reason is correct after a wakeup from deep sleep. We need to create a deep-sleep reset first and then check the reset reason. To support this, we can define multi-stage test cases, to group a set of test functions:

```

static void trigger_deepsleep(void)
{
    esp_sleep_enable_timer_wakeup(2000);
    esp_deep_sleep_start();
}

void check_deepsleep_reset_reason()
{
    RESET_REASON reason = rtc_get_reset_reason(0);
    TEST_ASSERT(reason == DEEPSLEEP_RESET);
}

TEST_CASE_MULTIPLE_STAGES("reset reason check for deepsleep", "[esp32]", trigger_
↪deepsleep, check_deepsleep_reset_reason);

```

Multi-stage test cases present a group of test functions to users. It needs user interactions (select cases and select different stages) to run the case.

5.32.4 Building Unit Test App

Follow the setup instructions in the top-level esp-idf README. Make sure that `IDF_PATH` environment variable is set to point to the path of esp-idf top-level directory.

Change into `tools/unit-test-app` directory to configure and build it:

- `idf.py menuconfig` - configure unit test app.
- `idf.py -T all build` - build unit test app with tests for each component having tests in the `test` subdirectory.
- `idf.py -T "xxx yyy" build` - build unit test app with tests for some space-separated specific components (For instance: `idf.py -T heap build` - build unit tests only for heap component directory).
- `idf.py -T all -E "xxx yyy" build` - build unit test app with all unit tests, except for unit tests of some components (For instance: `idf.py -T all -E "ulp mbedtls" build` - build all unit tests excludes `ulp` and `mbedtls` components).

When the build finishes, it will print instructions for flashing the chip. You can simply run `idf.py flash` to flash all build output.

You can also run `idf.py -T all flash` or `idf.py -T xxx flash` to build and flash. Everything needed will be rebuilt automatically before flashing.

Use `menuconfig` to set the serial port for flashing.

5.32.5 Running Unit Tests

After flashing reset the ESP32 and it will boot the unit test app.

When unit test app is idle, press “Enter” will make it print test menu with all available tests:

```
Here's the test menu, pick your combo:
(1)   "esp_ota_begin() verifies arguments" [ota]
(2)   "esp_ota_get_next_update_partition logic" [ota]
(3)   "Verify bootloader image in flash" [bootloader_support]
(4)   "Verify unit test app image" [bootloader_support]
(5)   "can use new and delete" [cxx]
(6)   "can call virtual functions" [cxx]
(7)   "can use static initializers for non-POD types" [cxx]
(8)   "can use std::vector" [cxx]
(9)   "static initialization guards work as expected" [cxx]
(10)  "global initializers run in the correct order" [cxx]
(11)  "before scheduler has started, static initializers work correctly" [cxx]
(12)  "adc2 work with wifi" [adc]
(13)  "gpio master/slave test example" [ignore][misc][test_env=UT_T2_1][multi_
->device]
      (1)   "gpio_master_test"
      (2)   "gpio_slave_test"
(14)  "SPI Master clockdiv calculation routines" [spi]
(15)  "SPI Master test" [spi][ignore]
(16)  "SPI Master test, interaction of multiple devs" [spi][ignore]
(17)  "SPI Master no response when switch from host1 (HSPI) to host2 (VSPI)"
->[spi]
(18)  "SPI Master DMA test, TX and RX in different regions" [spi]
(19)  "SPI Master DMA test: length, start, not aligned" [spi]
(20)  "reset reason check for deepsleep" [esp32][test_env=UT_T2_1][multi_stage]
      (1)   "trigger_deepsleep"
      (2)   "check_deepsleep_reset_reason"
```

The normal case will print the case name and description. Master-slave cases will also print the sub-menu (the registered test function names).

Test cases can be run by inputting one of the following:

- Test case name in quotation marks to run a single test case

- Test case index to run a single test case
- Module name in square brackets to run all test cases for a specific module
- An asterisk to run all test cases

[multi_device] and [multi_stage] tags tell the test runner whether a test case is a multiple devices or multiple stages of test case. These tags are automatically added by `TEST_CASE_MULTIPLE_STAGES` and `TEST_CASE_MULTIPLE_DEVICES` macros.

After you select a multi-device test case, it will print sub-menu:

```
Running gpio master/slave test example...
gpio master/slave test example
  (1)  "gpio_master_test"
  (2)  "gpio_slave_test"
```

You need to input a number to select the test running on the DUT.

Similar to multi-device test cases, multi-stage test cases will also print sub-menu:

```
Running reset reason check for deepsleep...
reset reason check for deepsleep
  (1)  "trigger_deepsleep"
  (2)  "check_deepsleep_reset_reason"
```

First time you execute this case, input 1 to run first stage (trigger deepsleep). After DUT is rebooted and able to run test cases, select this case again and input 2 to run the second stage. The case only passes if the last stage passes and all previous stages trigger reset.

5.32.6 Timing Code with Cache Compensated Timer

Instructions and data stored in external memory (e.g. SPI Flash and SPI RAM) are accessed through the CPU's unified instruction and data cache. When code or data is in cache, access is very fast (i.e., a cache hit).

However, if the instruction or data is not in cache, it needs to be fetched from external memory (i.e., a cache miss). Access to external memory is significantly slower, as the CPU must execute stall cycles whilst waiting for the instruction or data to be retrieved from external memory. This can cause the overall code execution speed to vary depending on the number of cache hits or misses.

Code and data placements can vary between builds, and some arrangements may be more favorable with regards to cache access (i.e., minimizing cache misses). This can technically affect execution speed, however these factors are usually irrelevant as their effect 'average out' over the device's operation.

The effect of the cache on execution speed, however, can be relevant in benchmarking scenarios (especially microbenchmarks). There might be some variability in measured time between runs and between different builds. A technique for eliminating for some of the variability is to place code and data in instruction or data RAM (IRAM/DRAM), respectively. The CPU can access IIRAM and DRAM directly, eliminating the cache out of the equation. However, this might not always be viable as the size of IIRAM and DRAM is limited.

The cache compensated timer is an alternative to placing the code/data to be benchmarked in IIRAM/DRAM. This timer uses the processor's internal event counters in order to determine the amount of time spent on waiting for code/data in case of a cache miss, then subtract that from the recorded wall time.

```
// Start the timer
ccomp_timer_start();

// Function to time
func_code_to_time();

// Stop the timer, and return the elapsed time in microseconds relative to
// ccomp_timer_start
int64_t t = ccomp_timer_stop();
```

One limitation of the cache compensated timer is that the task that benchmarked functions should be pinned to a core. This is due to each core having its own event counters that are independent of each other. For example, if `ccomp_timer_start` gets called on one core, put to sleep by the scheduler, wakes up, and gets rescheduled on the other core, then the corresponding `ccomp_timer_stop` will be invalid.

5.33 Wi-Fi Driver

5.33.1 ESP32 Wi-Fi Feature List

- Support Station-only mode, AP-only mode, Station/AP-coexistence mode
- Support IEEE-802.11B, IEEE-802.11G, IEEE802.11N and APIs to configure the protocol mode
- Support WPA/WPA2/WPA2-Enterprise and WPS
- Support AMPDU, HT40, QoS and other key features
- Support Modem-sleep
- Support an Espressif-specific protocol which, in turn, supports up to **1 km** of data traffic
- Up to 20 MBit/sec TCP throughput and 30 MBit/sec UDP throughput over the air
- Support Sniffer
- Support set fast_crypto algorithm and normal algorithm switch which used in wifi connect
- Support both fast scan and all channel scan feature
- Support multiple antennas
- Support channel state information

5.33.2 How To Write a Wi-Fi Application

Preparation

Generally, the most effective way to begin your own Wi-Fi application is to select an example which is similar to your own application, and port the useful part into your project. It is not a MUST but it is strongly recommended that you take some time to read this article first, especially if you want to program a robust Wi-Fi application. This article is supplementary to the Wi-Fi APIs/Examples. It describes the principles of using the Wi-Fi APIs, the limitations of the current Wi-Fi API implementation, and the most common pitfalls in using Wi-Fi. This article also reveals some design details of the Wi-Fi driver. We recommend that you become familiar at least with the following sections: [<ESP32 Wi-Fi API Error Code>](#), [<ESP32 Wi-Fi Programming Model>](#), and [<ESP32 Wi-Fi Event Description>](#).

Setting Wi-Fi Compile-time Options

Refer to [<Wi-Fi Menuconfig>](#)

Init Wi-Fi

Refer to [<ESP32 Wi-Fi Station General Scenario>](#), [<ESP32 Wi-Fi AP General Scenario>](#).

Start/Connect Wi-Fi

Refer to [<ESP32 Wi-Fi Station General Scenario>](#), [<ESP32 Wi-Fi AP General Scenario>](#).

Event-Handling

Generally, it is easy to write code in “sunny-day” scenarios, such as [<WIFI_EVENT_STA_START>](#), [<WIFI_EVENT_STA_CONNECTED>](#) etc. The hard part is to write routines in “rainy-day” scenarios, such as [<WIFI_EVENT_STA_DISCONNECTED>](#) etc. Good handling of “rainy-day” scenarios is fundamental to robust

Wi-Fi applications. Refer to [<ESP32 Wi-Fi Event Description>](#), [<ESP32 Wi-Fi Station General Scenario>](#), [<ESP32 Wi-Fi AP General Scenario>](#). See also [an overview of event handling in ESP-IDF](#).

Write Error-Recovery Routines Correctly at All Times

Just like the handling of “rainy-day” scenarios, a good error-recovery routine is also fundamental to robust Wi-Fi applications. Refer to [<ESP32 Wi-Fi API Error Code>](#)

5.33.3 ESP32 Wi-Fi API Error Code

All of the ESP32 Wi-Fi APIs have well-defined return values, namely, the error code. The error code can be categorized into

- No errors, e.g. ESP_OK means that the API returns successfully
- Recoverable errors, such as ESP_ERR_NO_MEM, etc.
- Non-recoverable, non-critical errors
- Non-recoverable, critical errors

Whether the error is critical or not depends on the API and the application scenario, and it is defined by the API user.

The primary principle to write a robust application with Wi-Fi API is to always check the error code and write the error-handling code. Generally, the error-handling code can be used:

- for recoverable errors, in which case you can write a recoverable-error code. For example, when `esp_wifi_start` returns `ESP_ERR_NO_MEM`, the recoverable-error code `vTaskDelay` can be called, in order to get a microsecond's delay for another try.
- for non-recoverable, yet non-critical, errors, in which case printing the error code is a good method for error handling.
- for non-recoverable, critical errors, in which case “assert” may be a good method for error handling. For example, if `esp_wifi_set_mode` returns `ESP_ERR_WIFI_NOT_INIT`, it means that the Wi-Fi driver is not initialized by `esp_wifi_init` successfully. You can detect this kind of error very quickly in the application development phase.

In `esp_err.h`, `ESP_ERROR_CHECK` checks the return values. It is a rather commonplace error-handling code and can be used as the default error-handling code in the application development phase. However, we strongly recommend that API users write their own error-handling code.

5.33.4 ESP32 Wi-Fi API Parameter Initialization

When initializing struct parameters for the API, one of two approaches should be followed: - explicitly set all fields of the parameter or - use `get API` to get current configuration first, then set application specific fields

Initializing or getting the entire structure is very important because most of the time the value 0 indicates the default value is used. More fields may be added to the struct in the future and initializing these to zero ensures the application will still work correctly after IDF is updated to a new release.

5.33.5 ESP32 Wi-Fi Programming Model

The ESP32 Wi-Fi programming model is depicted as follows:

The Wi-Fi driver can be considered a black box that knows nothing about high-layer code, such as the TCP/IP stack, application task, event task, etc. The application task (code) generally calls [Wi-Fi driver APIs](#) to initialize Wi-Fi and handles Wi-Fi events when necessary. Wi-Fi driver receives API calls, handles them, and post events to the application.

Wi-Fi event handling is based on the [esp_event library](#). Events are sent by the Wi-Fi driver to the [default event loop](#). Application may handle these events in callbacks registered using [esp_event_handler_register\(\)](#). Wi-Fi events are also handled by [esp_netif component](#) to provide a set of default behaviors. For example, when Wi-Fi station connects to an AP, `esp_netif` will automatically start the DHCP client (by default).

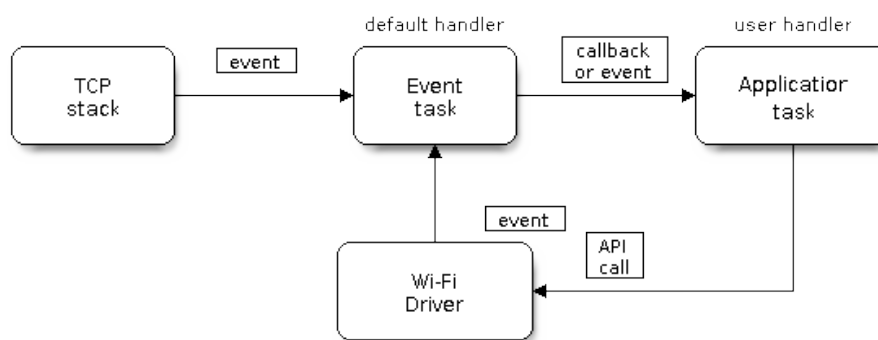


Fig. 54: Wi-Fi Programming Model

5.33.6 ESP32 Wi-Fi Event Description

WIFI_EVENT_WIFI_READY

The Wi-Fi driver will never generate this event, which, as a result, can be ignored by the application event callback. This event may be removed in future releases.

WIFI_EVENT_SCAN_DONE

The scan-done event is triggered by `esp_wifi_scan_start()` and will arise in the following scenarios:

- The scan is completed, e.g., the target AP is found successfully, or all channels have been scanned.
- The scan is stopped by `esp_wifi_scan_stop()`.
- The `esp_wifi_scan_start()` is called before the scan is completed. A new scan will override the current scan and a scan-done event will be generated.

The scan-done event will not arise in the following scenarios:

- It is a blocked scan.
- The scan is caused by `esp_wifi_connect()`.

Upon receiving this event, the event task does nothing. The application event callback needs to call `esp_wifi_scan_get_ap_num()` and `esp_wifi_scan_get_ap_records()` to fetch the scanned AP list and trigger the Wi-Fi driver to free the internal memory which is allocated during the scan (**do not forget to do this!**)! Refer to ‘ESP32 Wi-Fi Scan’ for a more detailed description.

WIFI_EVENT_STA_START

If `esp_wifi_start()` returns `ESP_OK` and the current Wi-Fi mode is Station or AP+Station, then this event will arise. Upon receiving this event, the event task will initialize the LwIP network interface (netif). Generally, the application event callback needs to call `esp_wifi_connect()` to connect to the configured AP.

WIFI_EVENT_STA_STOP

If `esp_wifi_stop()` returns `ESP_OK` and the current Wi-Fi mode is Station or AP+Station, then this event will arise. Upon receiving this event, the event task will release the station’s IP address, stop the DHCP client, remove TCP/UDP-related connections and clear the LwIP station netif, etc. The application event callback generally does not need to do anything.

WIFI_EVENT_STA_CONNECTED

If `esp_wifi_connect()` returns `ESP_OK` and the station successfully connects to the target AP, the connection event will arise. Upon receiving this event, the event task starts the DHCP client and begins the DHCP process of getting the IP address. Then, the Wi-Fi driver is ready for sending and receiving data. This moment is good for beginning the application work, provided that the application does not depend on LwIP, namely the IP address. However, if the application is LwIP-based, then you need to wait until the *got ip* event comes in.

WIFI_EVENT_STA_DISCONNECTED

This event can be generated in the following scenarios:

- When `esp_wifi_disconnect()`, or `esp_wifi_stop()`, or `esp_wifi_deinit()` is called and the station is already connected to the AP.
- When `esp_wifi_connect()` is called, but the Wi-Fi driver fails to set up a connection with the AP due to certain reasons, e.g. the scan fails to find the target AP, authentication times out, etc. If there are more than one AP with the same SSID, the disconnected event is raised after the station fails to connect all of the found APs.
- When the Wi-Fi connection is disrupted because of specific reasons, e.g., the station continuously loses N beacons, the AP kicks off the station, the AP's authentication mode is changed, etc.

Upon receiving this event, the default behavior of the event task is: - Shuts down the station's LwIP netif. - Notifies the LwIP task to clear the UDP/TCP connections which cause the wrong status to all sockets. For socket-based applications, the application callback can choose to close all sockets and re-create them, if necessary, upon receiving this event.

The most common event handle code for this event in application is to call `esp_wifi_connect()` to reconnect the Wi-Fi. However, if the event is raised because `esp_wifi_disconnect()` is called, the application should not call `esp_wifi_connect()` to reconnect. It's application's responsibility to distinguish whether the event is caused by `esp_wifi_disconnect()` or other reasons. Sometimes a better reconnect strategy is required, refer to [<Wi-Fi Reconnect>](#) and [<Scan When Wi-Fi Is Connecting>](#).

Another thing deserves our attention is that the default behavior of LwIP is to abort all TCP socket connections on receiving the disconnect. Most of time it is not a problem. However, for some special application, this may not be what they want, consider following scenarios:

- The application creates a TCP connection to maintain the application-level keep-alive data that is sent out every 60 seconds.
- Due to certain reasons, the Wi-Fi connection is cut off, and the `<WIFI_EVENT_STA_DISCONNECTED>` is raised. According to the current implementation, all TCP connections will be removed and the keep-alive socket will be in a wrong status. However, since the application designer believes that the network layer should NOT care about this error at the Wi-Fi layer, the application does not close the socket.
- Five seconds later, the Wi-Fi connection is restored because `esp_wifi_connect()` is called in the application event callback function. **Moreover, the station connects to the same AP and gets the same IPV4 address as before.**
- Sixty seconds later, when the application sends out data with the keep-alive socket, the socket returns an error and the application closes the socket and re-creates it when necessary.

In above scenario, ideally, the application sockets and the network layer should not be affected, since the Wi-Fi connection only fails temporarily and recovers very quickly. The application can enable "Keep TCP connections when IP changed" via LwIP menuconfig.

IP_EVENT_STA_GOT_IP

This event arises when the DHCP client successfully gets the IPV4 address from the DHCP server, or when the IPV4 address is changed. The event means that everything is ready and the application can begin its tasks (e.g., creating sockets).

The IPV4 may be changed because of the following reasons:

- The DHCP client fails to renew/rebind the IPV4 address, and the station's IPV4 is reset to 0.
- The DHCP client rebinds to a different address.
- The static-configured IPV4 address is changed.

Whether the IPV4 address is changed or NOT is indicated by field `ip_change` of `ip_event_got_ip_t`.

The socket is based on the IPV4 address, which means that, if the IPV4 changes, all sockets relating to this IPV4 will become abnormal. Upon receiving this event, the application needs to close all sockets and recreate the application when the IPV4 changes to a valid one.

IP_EVENT_GOT_IP6

This event arises when the IPV6 SLAAC support auto-configures an address for the ESP32, or when this address changes. The event means that everything is ready and the application can begin its tasks (e.g., creating sockets).

IP_STA_LOST_IP

This event arises when the IPV4 address become invalid.

`IP_STA_LOST_IP` doesn't arise immediately after the WiFi disconnects, instead it starts an IPV4 address lost timer, if the IPV4 address is got before ip lost timer expires, `IP_EVENT_STA_LOST_IP` doesn't happen. Otherwise, the event arises when IPV4 address lost timer expires.

Generally the application don't need to care about this event, it is just a debug event to let the application know that the IPV4 address is lost.

WIFI_EVENT_AP_START

Similar to `<WIFI_EVENT_STA_START>`.

WIFI_EVENT_AP_STOP

Similar to `<WIFI_EVENT_STA_STOP>`.

WIFI_EVENT_AP_STACONNECTED

Every time a station is connected to ESP32 AP, the `<WIFI_EVENT_AP_STACONNECTED>` will arise. Upon receiving this event, the event task will do nothing, and the application callback can also ignore it. However, you may want to do something, for example, to get the info of the connected STA, etc.

WIFI_EVENT_AP_STADISCONNECTED

This event can happen in the following scenarios:

- The application calls `esp_wifi_disconnect()`, or `esp_wifi_deinit_sta()`, to manually disconnect the station.
- The Wi-Fi driver kicks off the station, e.g. because the AP has not received any packets in the past five minutes, etc. The time can be modified by `esp_wifi_set_inactive_time()`.
- The station kicks off the AP.

When this event happens, the event task will do nothing, but the application event callback needs to do something, e.g., close the socket which is related to this station, etc.

WIFI_EVENT_AP_PROBEREQRECVED

This event is disabled by default. The application can enable it via API `esp_wifi_set_event_mask()`. When this event is enabled, it will be raised each time the AP receives a probe request.

5.33.7 ESP32 Wi-Fi Station General Scenario

Below is a “big scenario” which describes some small scenarios in Station mode:

1. Wi-Fi/LwIP Init Phase

- s1.1: The main task calls `esp_netif_init()` to create an LwIP core task and initialize LwIP-related work.
- s1.2: The main task calls `esp_event_loop_init()` to create a system Event task and initialize an application event's callback function. In the scenario above, the application event's callback function does nothing but relaying the event to the application task.
- s1.3: The main task calls `esp_netif_create_default_wifi_ap()` or `esp_netif_create_default_wifi_sta()` to create default network interface instance binding station or AP with TCP/IP stack.
- s1.4: The main task calls `esp_wifi_init()` to create the Wi-Fi driver task and initialize the Wi-Fi driver.
- s1.5: The main task calls OS API to create the application task.

Step 1.1~1.5 is a recommended sequence that initializes a Wi-Fi/LwIP-based application. However, it is **NOT** a must-follow sequence, which means that you can create the application task in step 1.1 and put all other initializations in the application task. Moreover, you may not want to create the application task in the initialization phase if the application task depends on the sockets. Rather, you can defer the task creation until the IP is obtained.

2. Wi-Fi Configuration Phase

Once the Wi-Fi driver is initialized, you can start configuring the Wi-Fi driver. In this scenario, the mode is Station, so you may need to call `esp_wifi_set_mode(WIFI_MODE_STA)` to configure the Wi-Fi mode as Station. You can call other `esp_wifi_set_xxx` APIs to configure more settings, such as the protocol mode, country code, bandwidth, etc. Refer to [<ESP32 Wi-Fi Configuration>](#).

Generally, we configure the Wi-Fi driver before setting up the Wi-Fi connection, but this is **NOT** mandatory, which means that you can configure the Wi-Fi connection anytime, provided that the Wi-Fi driver is initialized successfully. However, if the configuration does not need to change after the Wi-Fi connection is set up, you should configure the Wi-Fi driver at this stage, because the configuration APIs (such as `esp_wifi_set_protocol`) will cause the Wi-Fi to reconnect, which may not be desirable.

If the Wi-Fi NVS flash is enabled by menuconfig, all Wi-Fi configuration in this phase, or later phases, will be stored into flash. When the board powers on/reboots, you do not need to configure the Wi-Fi driver from scratch. You only need to call `esp_wifi_get_xxx` APIs to fetch the configuration stored in flash previously. You can also configure the Wi-Fi driver if the previous configuration is not what you want.

3. Wi-Fi Start Phase

- s3.1: Call `esp_wifi_start` to start the Wi-Fi driver.
- s3.2: The Wi-Fi driver posts [<WIFI_EVENT_STA_START>](#) to the event task; then, the event task will do some common things and will call the application event callback function.
- s3.3: The application event callback function relays the [<WIFI_EVENT_STA_START>](#) to the application task. We recommend that you call `esp_wifi_connect()`. However, you can also call `esp_wifi_connect()` in other phrases after the [<WIFI_EVENT_STA_START>](#) arises.

4. Wi-Fi Connect Phase

- s4.1: Once `esp_wifi_connect()` is called, the Wi-Fi driver will start the internal scan/connection process.

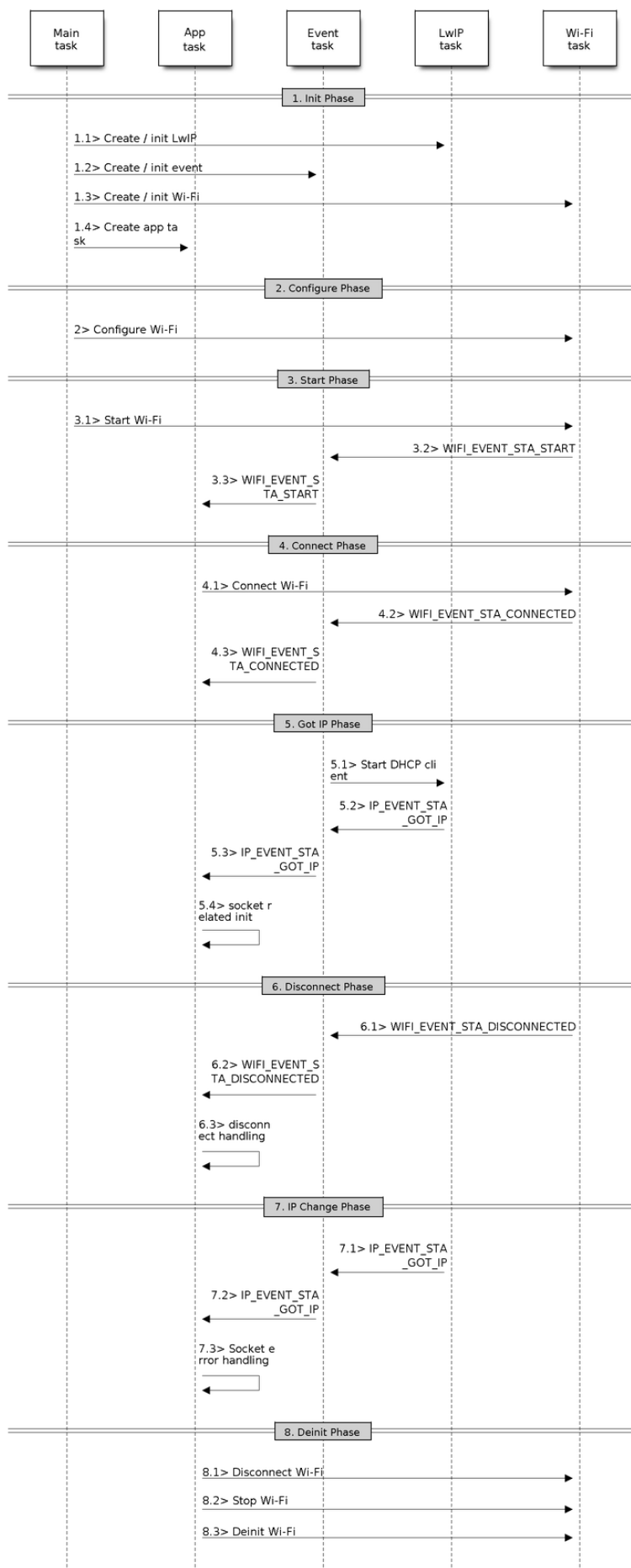


Fig. 55: Sample Wi-Fi Event Scenarios in Station Mode

- s4.2: If the internal scan/connection process is successful, the `<WIFI_EVENT_STA_CONNECTED>` will be generated. In the event task, it starts the DHCP client, which will finally trigger the DHCP process.
- s4.3: In the above-mentioned scenario, the application event callback will relay the event to the application task. Generally, the application needs to do nothing, and you can do whatever you want, e.g., print a log, etc.

In step 4.2, the Wi-Fi connection may fail because, for example, the password is wrong, the AP is not found, etc. In a case like this, `<WIFI_EVENT_STA_DISCONNECTED>` will arise and the reason for such a failure will be provided. For handling events that disrupt Wi-Fi connection, please refer to phase 6.

5. Wi-Fi ‘Got IP’ Phase

- s5.1: Once the DHCP client is initialized in step 4.2, the *got IP* phase will begin.
- s5.2: If the IP address is successfully received from the DHCP server, then `<IP_EVENT_STA_GOT_IP>` will arise and the event task will perform common handling.
- s5.3: In the application event callback, `<IP_EVENT_STA_GOT_IP>` is relayed to the application task. For LwIP-based applications, this event is very special and means that everything is ready for the application to begin its tasks, e.g. creating the TCP/UDP socket, etc. A very common mistake is to initialize the socket before `<IP_EVENT_STA_GOT_IP>` is received. **DO NOT start the socket-related work before the IP is received.**

6. Wi-Fi Disconnect Phase

- s6.1: When the Wi-Fi connection is disrupted, e.g. because the AP is powered off, the RSSI is poor, etc., `<WIFI_EVENT_STA_DISCONNECTED>` will arise. This event may also arise in phase 3. Here, the event task will notify the LwIP task to clear/remove all UDP/TCP connections. Then, all application sockets will be in a wrong status. In other words, no socket can work properly when this event happens.
- s6.2: In the scenario described above, the application event callback function relays `<WIFI_EVENT_STA_DISCONNECTED>` to the application task. We recommend that `esp_wifi_connect()` be called to reconnect the Wi-Fi, close all sockets and re-create them if necessary. Refer to `<WIFI_EVENT_STA_DISCONNECTED>`.

7. Wi-Fi IP Change Phase

- s7.1: If the IP address is changed, the `<IP_EVENT_STA_GOT_IP>` will arise with “ip_change” set to true.
- s7.2: **This event is important to the application. When it occurs, the timing is good for closing all created sockets and recreating them.**

8. Wi-Fi Deinit Phase

- s8.1: Call `esp_wifi_disconnect()` to disconnect the Wi-Fi connectivity.
- s8.2: Call `esp_wifi_stop()` to stop the Wi-Fi driver.
- s8.3: Call `esp_wifi_deinit()` to unload the Wi-Fi driver.

5.33.8 ESP32 Wi-Fi AP General Scenario

Below is a “big scenario” which describes some small scenarios in AP mode:

5.33.9 ESP32 Wi-Fi Scan

Currently, the `esp_wifi_scan_start()` API is supported only in Station or Station+AP mode.

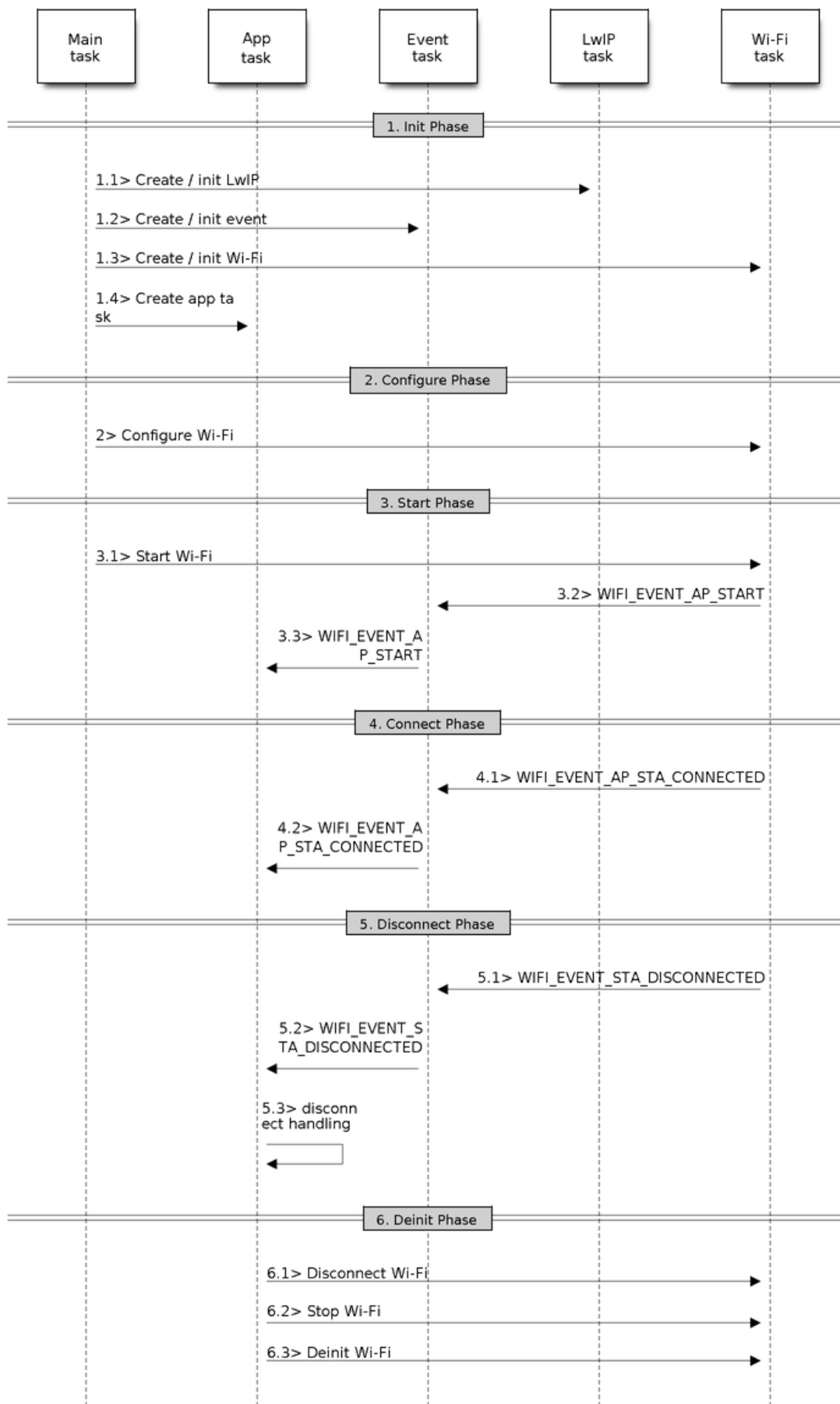


Fig. 56: Sample Wi-Fi Event Scenarios in AP Mode

Scan Type

| Mode | Description |
|------------------------------|---|
| Active Scan | Scan by sending a probe request. The default scan is an active scan. |
| Passive Scan | No probe request is sent out. Just switch to the specific channel and wait for a beacon. Application can enable it via the <code>scan_type</code> field of <code>wifi_scan_config_t</code> . |
| Foreground Scan | This scan is applicable when there is no Wi-Fi connection in Station mode. Foreground or background scanning is controlled by the Wi-Fi driver and cannot be configured by the application. |
| Background Scan | This scan is applicable when there is a Wi-Fi connection in Station mode or in Station+AP mode. Whether it is a foreground scan or background scan depends on the Wi-Fi driver and cannot be configured by the application. |
| All-Channel Scan | It scans all of the channels. If the <code>channel</code> field of <code>wifi_scan_config_t</code> is set to 0, it is an all-channel scan. |
| Specific Channel Scan | It scans specific channels only. If the <code>channel</code> field of <code>wifi_scan_config_t</code> set to 1, it is a specific-channel scan. |

The scan modes in above table can be combined arbitrarily, so we totally have 8 different scans:

- All-Channel Background Active Scan
- All-Channel Background Passive Scan
- All-Channel Foreground Active Scan
- All-Channel Foreground Passive Scan
- Specific-Channel Background Active Scan
- Specific-Channel Background Passive Scan
- Specific-Channel Foreground Active Scan
- Specific-Channel Foreground Passive Scan

Scan Configuration

The scan type and other per-scan attributes are configured by `esp_wifi_scan_start`. The table below provides a detailed description of `wifi_scan_config_t`.

| Field | Description |
|-------------|---|
| ssid | If the SSID is not NULL, it is only the AP with the same SSID that can be scanned. |
| bssid | If the BSSID is not NULL, it is only the AP with the same BSSID that can be scanned. |
| channel | If “channel” is 0, there will be an all-channel scan; otherwise, there will be a specific-channel scan. |
| show_hidden | If “show_hidden” is 0, the scan ignores the AP with a hidden SSID; otherwise, the scan considers the hidden AP a normal one. |
| scan_type | If “scan_type” is WIFI_SCAN_TYPE_ACTIVE, the scan is “active” ; otherwise, it is a “passive” one. |
| scan_time | <p>This field is used to control how long the scan dwells on each channel.</p> <p>For passive scans, scan_time.passive designates the dwell time for each channel.</p> <p>For active scans, dwell times for each channel are listed in the table below. Here, min is short for scan_time.active.min and max is short for scan_time.active.max.</p> <ul style="list-style-type: none"> • min=0, max=0: scan dwells on each channel for 120 ms. • min>0, max=0: scan dwells on each channel for 120 ms. • min=0, max>0: scan dwells on each channel for max ms. • min>0, max>0: the minimum time the scan dwells on each channel is min ms. If no AP is found during this time frame, the scan switches to the next channel. Otherwise, the scan dwells on the channel for max ms. <p>If you want to improve the performance of the the scan, you can try to modify these two parameters.</p> |

There also some global scan attributes which is configured by API esp_wifi_set_config, refer to [Station Basic Configuration](#)

Scan All APs In All Channels(foreground)

Scenario:

The scenario above describes an all-channel, foreground scan. The foreground scan can only occur in Station mode where the station does not connect to any AP. Whether it is a foreground or background scan is totally determined by the Wi-Fi driver, and cannot be configured by the application.

Detailed scenario description:

Scan Configuration Phase

- s1.1: Call esp_wifi_set_country() to set the country info if the default country info is not what you want, refer to [<Wi-Fi Country Code>](#).
- s1.2: Call esp_wifi_scan_start() to configure the scan. To do so, you can refer to [<Scan Configuration>](#). Since this is an all-channel scan, just set the SSID/BSSID/channel to 0.

Wi-Fi Driver' s Internal Scan Phase

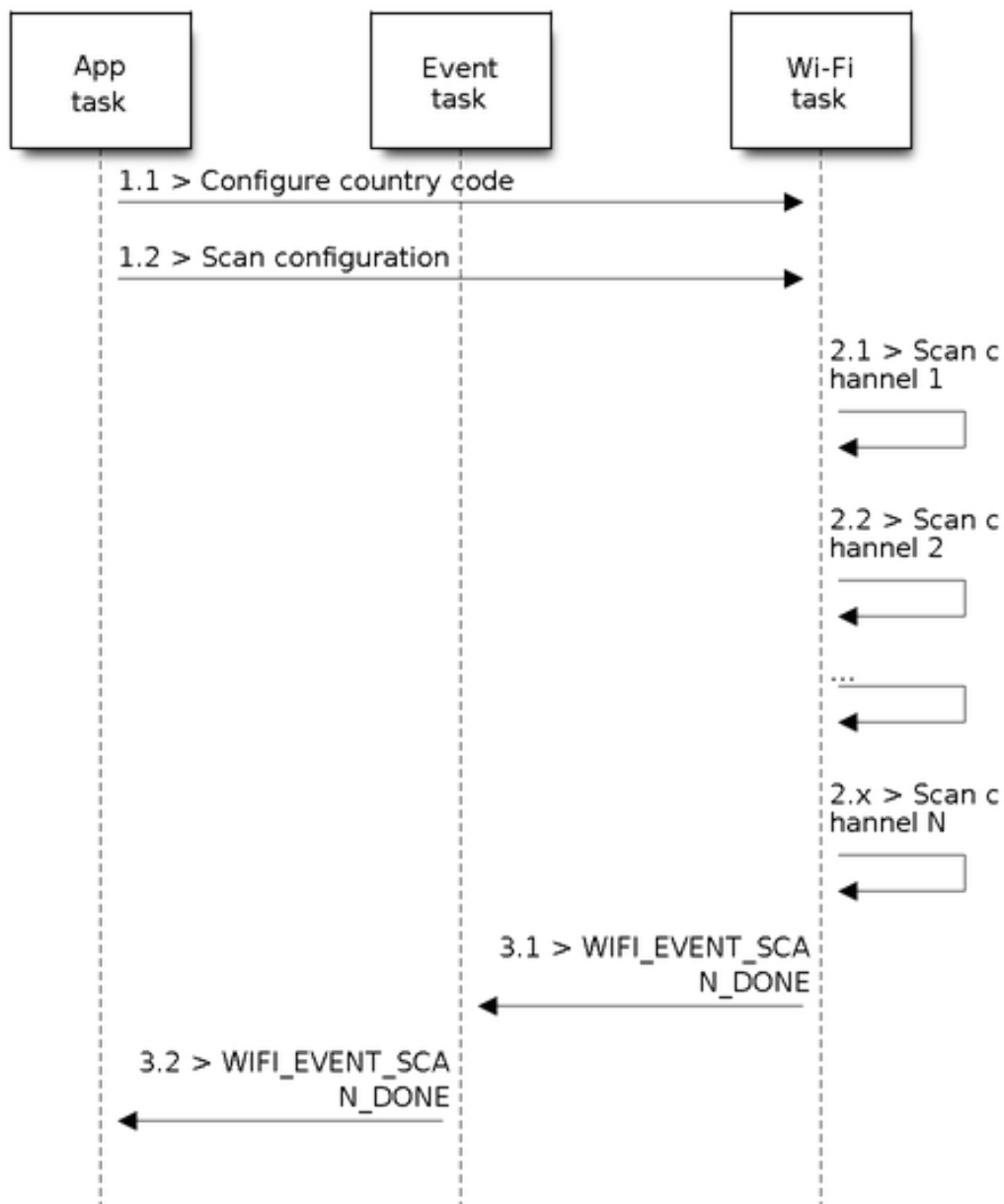


Fig. 57: Foreground Scan of all Wi-Fi Channels

- s2.1: The Wi-Fi driver switches to channel 1, in case the scan type is `WIFI_SCAN_TYPE_ACTIVE`, and broadcasts a probe request. Otherwise, the Wi-Fi will wait for a beacon from the APs. The Wi-Fi driver will stay in channel 1 for some time. The dwell time is configured in min/max time, with default value being 120 ms.
- s2.2: The Wi-Fi driver switches to channel 2 and performs the same operation as in step 2.1.
- s2.3: The Wi-Fi driver scans the last channel N, where N is determined by the country code which is configured in step 1.1.

Scan-Done Event Handling Phase

- s3.1: When all channels are scanned, `<WIFI_EVENT_SCAN_DONE>` will arise.
- s3.2: The application's event callback function notifies the application task that `<WIFI_EVENT_SCAN_DONE>` is received. `esp_wifi_scan_get_ap_num()` is called to get the number of APs that have been found in this scan. Then, it allocates enough entries and calls `esp_wifi_scan_get_ap_records()` to get the AP records. Please note that the AP records in the Wi-Fi driver will be freed, once `esp_wifi_scan_get_ap_records()` is called. Do not call `esp_wifi_scan_get_ap_records()` twice for a single scan-done event. If `esp_wifi_scan_get_ap_records()` is not called when the scan-done event occurs, the AP records allocated by the Wi-Fi driver will not be freed. So, make sure you call `esp_wifi_scan_get_ap_records()`, yet only once.

Scan All APs on All Channels(background)

Scenario:

The scenario above is an all-channel background scan. Compared to *Scan All APs In All Channels(foreground)*, the difference in the all-channel background scan is that the Wi-Fi driver will scan the back-to-home channel for 30 ms before it switches to the next channel to give the Wi-Fi connection a chance to transmit/receive data.

Scan for a Specific AP in All Channels

Scenario:

This scan is similar to *Scan All APs In All Channels(foreground)*. The differences are:

- s1.1: In step 1.2, the target AP will be configured to SSID/BSSID.
- s2.1~s2.N: Each time the Wi-Fi driver scans an AP, it will check whether it is a target AP or not. If the scan is `WIFI_FAST_SCAN` scan and the target AP is found, then the scan-done event will arise and scanning will end; otherwise, the scan will continue. Please note that the first scanned channel may not be channel 1, because the Wi-Fi driver optimizes the scanning sequence.

If there are multiple APs which match the target AP info, for example, if we happen to scan two APs whose SSID is "ap". If the scan is `WIFI_FAST_SCAN`, then only the first scanned "ap" will be found, if the scan is `WIFI_ALL_CHANNEL_SCAN`, both "ap" will be found and the station will connect the "ap" according to the configured strategy, refer to *Station Basic Configuration*.

You can scan a specific AP, or all of them, in any given channel. These two scenarios are very similar.

Scan in Wi-Fi Connect

When `esp_wifi_connect()` is called, then the Wi-Fi driver will try to scan the configured AP first. The scan in "Wi-Fi Connect" is the same as *Scan for a Specific AP In All Channels*, except that no scan-done event will be generated when the scan is completed. If the target AP is found, then the Wi-Fi driver will start the Wi-Fi connection; otherwise, `<WIFI_EVENT_STA_DISCONNECTED>` will be generated. Refer to *Scan for a Specific AP in All Channels*

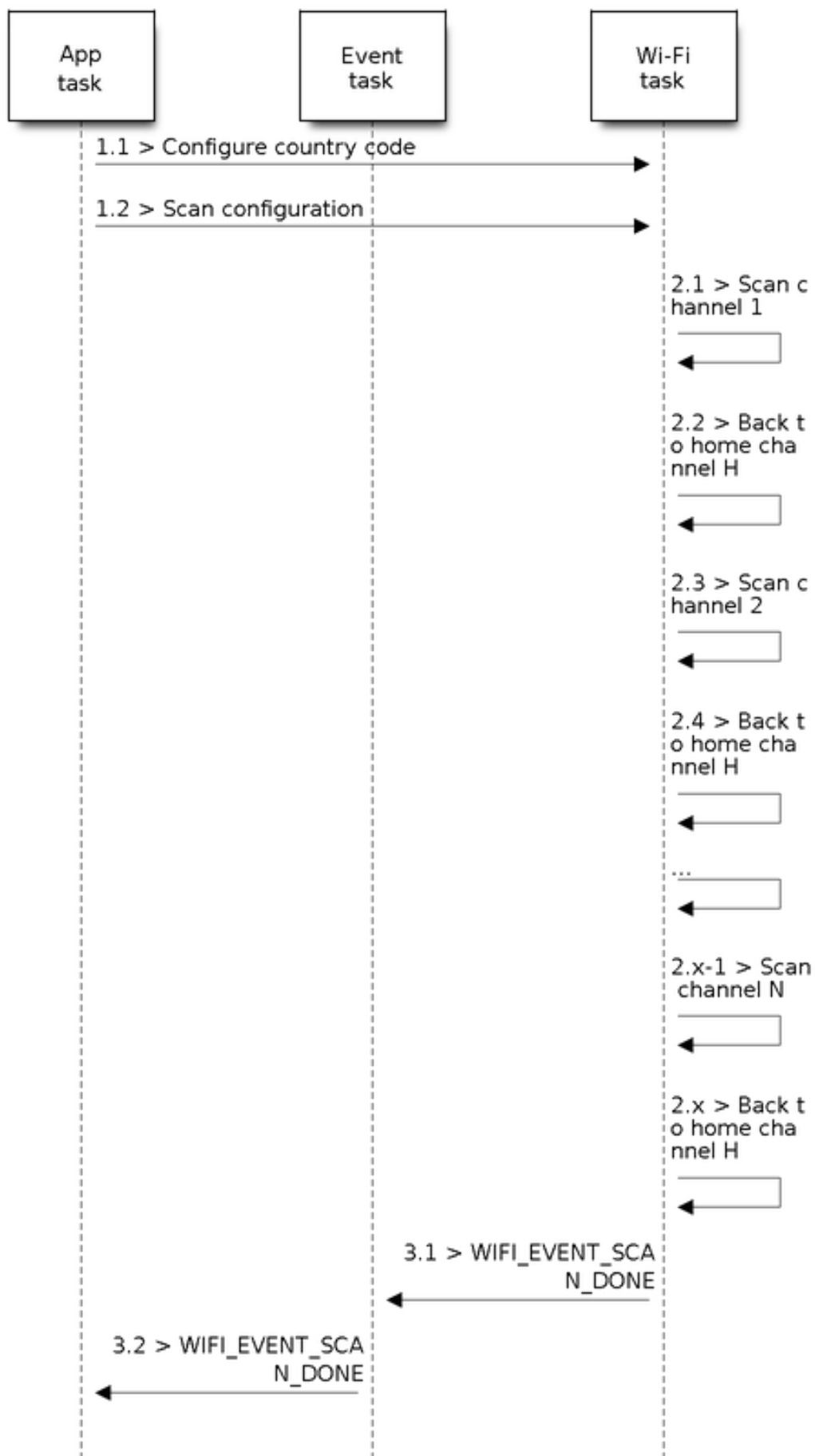


Fig. 58: Background Scan of all Wi-Fi Channels

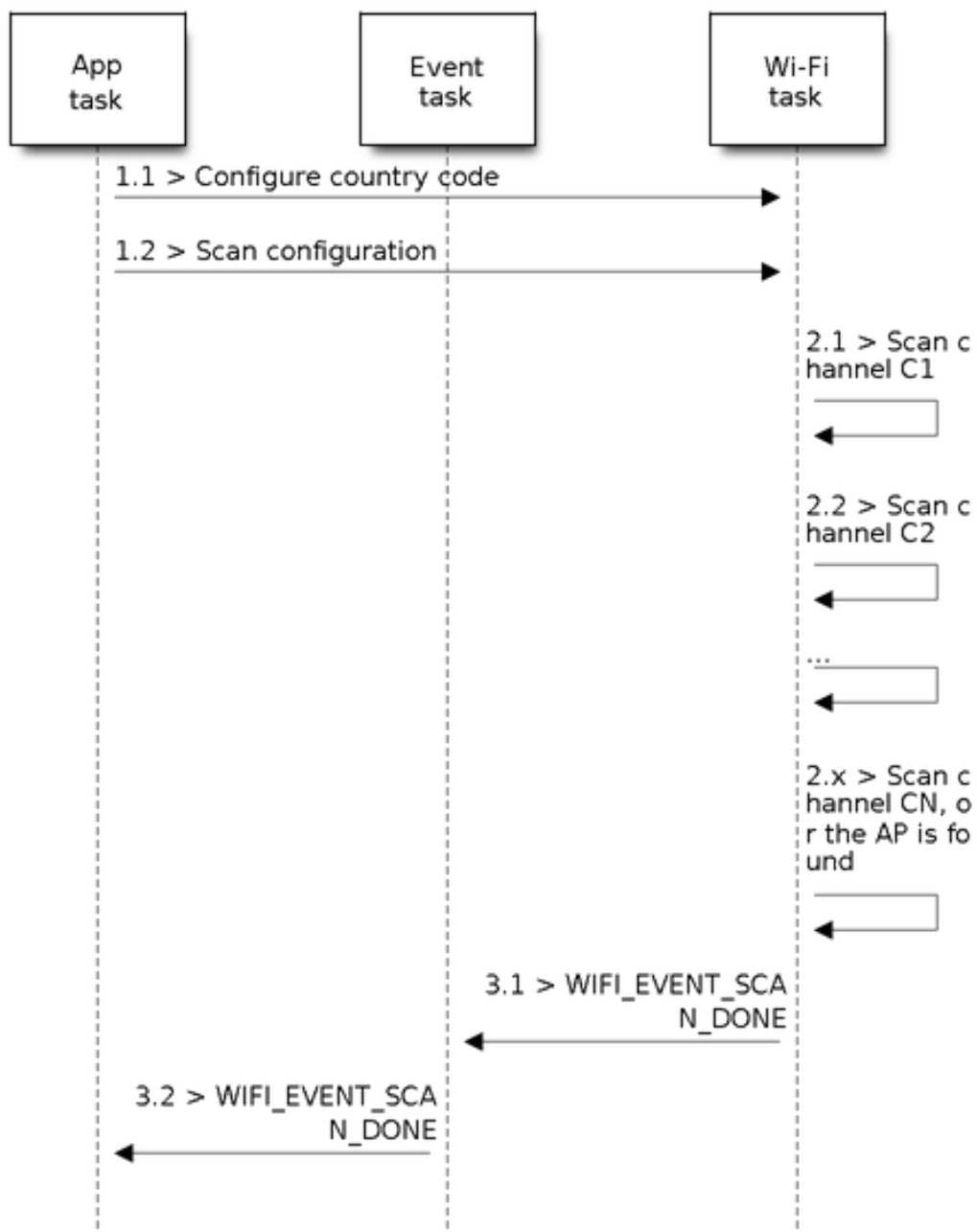


Fig. 59: Scan of specific Wi-Fi Channels

Scan In Blocked Mode

If the block parameter of `esp_wifi_scan_start()` is true, then the scan is a blocked one, and the application task will be blocked until the scan is done. The blocked scan is similar to an unblocked one, except that no scan-done event will arise when the blocked scan is completed.

Parallel Scan

Two application tasks may call `esp_wifi_scan_start()` at the same time, or the same application task calls `esp_wifi_scan_start()` before it gets a scan-done event. Both scenarios can happen. **However, the Wi-Fi driver does not support multiple concurrent scans adequately. As a result, concurrent scans should be avoided.** Support for concurrent scan will be enhanced in future releases, as the ESP32's Wi-Fi functionality improves continuously.

Scan When Wi-Fi Is Connecting

The `esp_wifi_scan_start()` fails immediately if the Wi-Fi is in connecting process because the connecting has higher priority than the scan. If scan fails because of connecting, the recommended strategy is to delay sometime and retry scan again, the scan will succeed once the connecting is completed.

However, the retry/delay strategy may not work all the time. Considering following scenario: - The station is connecting a non-existed AP or if the station connects the existed AP with a wrong password, it always raises the event `<WIFI_EVENT_STA_DISCONNECTED>`. - The application call `esp_wifi_connect()` to do reconnection on receiving the disconnect event. - Another application task, e.g. the console task, call `esp_wifi_scan_start()` to do scan, the scan always fails immediately because the station is keeping connecting. - When scan fails, the application simply delay sometime and retry the scan.

In above scenario the scan will never succeed because the connecting is in process. So if the application supports similar scenario, it needs to implement a better reconnect strategy. E.g. - The application can choose to define a maximum continuous reconnect counter, stop reconnect once the reconnect reaches the max counter. - The application can choose to do reconnect immediately in the first N continuous reconnect, then give a delay sometime and reconnect again.

The application can define its own reconnect strategy to avoid the scan starve to death. Refer to `<Wi-Fi Reconnect>`.

5.33.10 ESP32 Wi-Fi Station Connecting Scenario

This scenario only depicts the case when there is only one target AP are found in scan phase, for the scenario that more than one AP with the same SSID are found, refer to `<ESP32 Wi-Fi Station Connecting When Multiple APs Are Found>`.

Generally, the application does not need to care about the connecting process. Below is a brief introduction to the process for those who are really interested.

Scenario:

Scan Phase

- s1.1, The Wi-Fi driver begins scanning in “Wi-Fi Connect” . Refer to `<Scan in Wi-Fi Connect>` for more details.
- s1.2, If the scan fails to find the target AP, `<WIFI_EVENT_STA_DISCONNECTED>` will arise and the reason-code will be `WIFI_REASON_NO_AP_FOUND`. Refer to `<Wi-Fi Reason Code>`.

Auth Phase

- s2.1, The authentication request packet is sent and the auth timer is enabled.

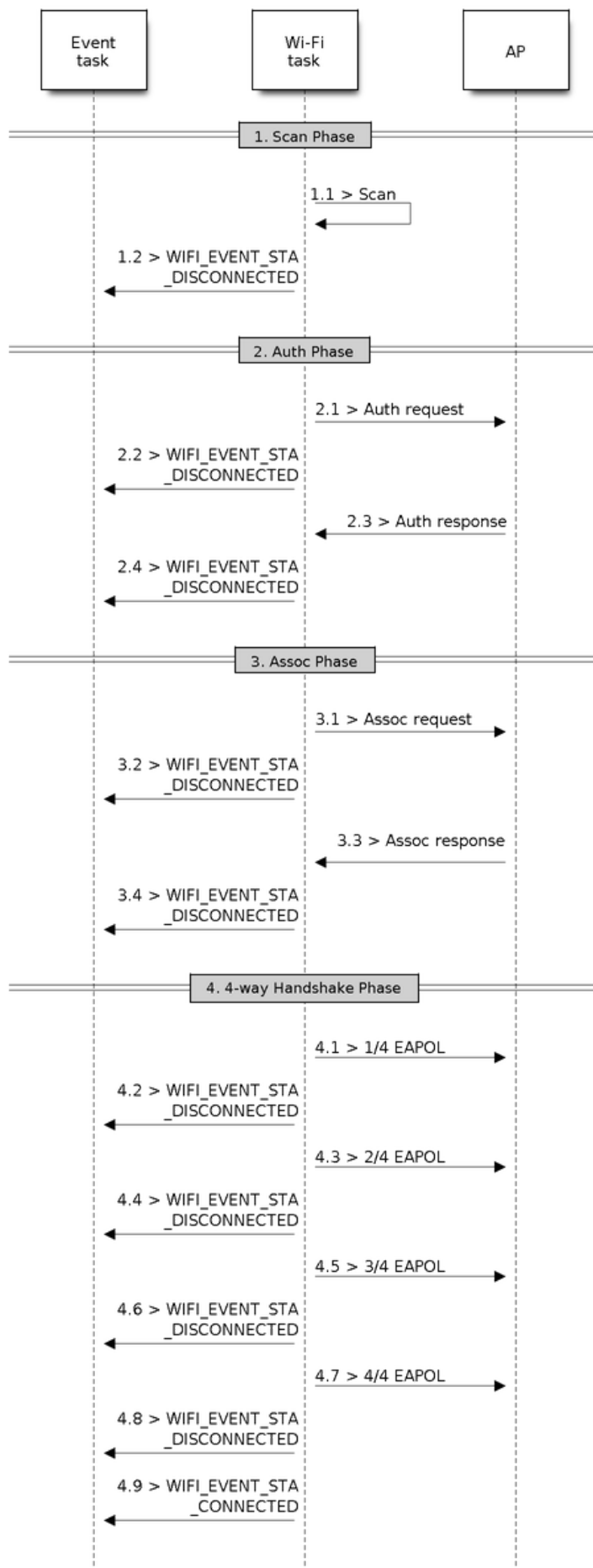


Fig. 60: Wi-Fi Station Connecting Process

- s2.2, If the authentication response packet is not received before the authentication timer times out, `<WIFI_EVENT_STA_DISCONNECTED>` will arise and the reason-code will be `WIFI_REASON_AUTH_EXPIRE`. Refer to [<Wi-Fi Reason Code>](#).
- s2.3, The auth-response packet is received and the auth-timer is stopped.
- s2.4, The AP rejects authentication in the response and `<WIFI_EVENT_STA_DISCONNECTED>` arises, while the reason-code is `WIFI_REASON_AUTH_FAIL` or the reasons specified by the AP. Refer to [<Wi-Fi Reason Code>](#).

Association Phase

- s3.1, The association request is sent and the association timer is enabled.
- s3.2, If the association response is not received before the association timer times out, `<WIFI_EVENT_STA_DISCONNECTED>` will arise and the reason-code will be `WIFI_REASON_ASSOC_EXPIRE`. Refer to [<Wi-Fi Reason Code>](#).
- s3.3, The association response is received and the association timer is stopped.
- s3.4, The AP rejects the association in the response and `<WIFI_EVENT_STA_DISCONNECTED>` arises, while the reason-code is the one specified in the association response. Refer to [<Wi-Fi Reason Code>](#).

Four-way Handshake Phase

- s4.1, The four-way handshake is sent out and the association timer is enabled.
- s4.2, If the association response is not received before the association timer times out, `<WIFI_EVENT_STA_DISCONNECTED>` will arise and the reason-code will be `WIFI_REASON_ASSOC_EXPIRE`. Refer to [<Wi-Fi Reason Code>](#).
- s4.3, The association response is received and the association timer is stopped.
- s4.4, The AP rejects the association in the response and `<WIFI_EVENT_STA_DISCONNECTED>` arises and the reason-code will be the one specified in the association response. Refer to [<Wi-Fi Reason Code>](#).

Wi-Fi Reason Code

The table below shows the reason-code defined in ESP32. The first column is the macro name defined in `esp_wifi_types.h`. The common prefix `WIFI_REASON` is removed, which means that `UNSPECIFIED` actually stands for `WIFI_REASON_UNSPECIFIED` and so on. The second column is the value of the reason. The third column is the standard value to which this reason is mapped in section 8.4.1.7 of `ieee802.11-2012`. (For more information, refer to the standard mentioned above.) The last column is a description of the reason.

| Reason code | Value | Mapped To | Description |
|-------------------|-------|--|---|
| UNSPECIFIED | 1 | 1 | Generally, it means an internal failure, e.g., the memory runs out, the internal TX fails, or the reason is received from the remote side, etc. |
| AUTH_EXPIRE | 2 | 2 | <p>The previous authentication is no longer valid.</p> <p>For the ESP32 Station, this reason is reported when:</p> <ul style="list-style-type: none"> • auth is timed out • the reason is received from the AP. <p>For the ESP32 AP, this reason is reported when:</p> <ul style="list-style-type: none"> • the AP has not received any packets from the station in the past five minutes. • the AP is stopped by calling <code>esp_wifi_stop()</code>. • the station is deauthenticated by calling <code>esp_wifi_deauth_sta()</code> |
| AUTH_LEAVE | 3 | 3 | <p>De-authenticated, because the sending STA is leaving (or has left).</p> <p>For the ESP32 Station, this reason is reported when:</p> <ul style="list-style-type: none"> • it is received from the AP. |
| ASSOC_EXPIRE | 4 | 4 | <p>Disassociated due to inactivity.</p> <p>For the ESP32 Station, this reason is reported when:</p> <ul style="list-style-type: none"> • it is received from the AP. <p>For the ESP32 AP, this reason is reported when:</p> <ul style="list-style-type: none"> • the AP has not received any packets from the station in the past five minutes. • the AP is stopped by calling <code>esp_wifi_stop()</code>. • the station is deauthenticated by calling <code>esp_wifi_deauth_sta()</code> |
| Espressif Systems | | 1707 | Release v4.1.3 |
| ASSOC_TOOMANY | 5 | Submit Document Feedback | Disassociated, because the AP is unable to handle all currently associated STAs at the same time. |

5.33.11 ESP32 Wi-Fi Station Connecting When Multiple APs Are Found

This scenario is similar as [<ESP32 Wi-Fi Station Connecting Scenario>](#), the difference is the station will not raise the event [<WIFI_EVENT_STA_DISCONNECTED>](#) unless it fails to connect all of the found APs.

5.33.12 Wi-Fi Reconnect

The station may disconnect due to many reasons, e.g. the connected AP is restarted etc. It's the application's responsibility to do the reconnect. The recommended reconnect strategy is to call `esp_wifi_connect()` on receiving event [<WIFI_EVENT_STA_DISCONNECTED>](#).

Sometimes the application needs more complex reconnect strategy: - If the disconnect event is raised because the `esp_wifi_disconnect()` is called, the application may not want to do reconnect. - If the `esp_wifi_scan_start()` may be called at anytime, a better reconnect strategy is necessary, refer to [<Scan When Wi-Fi Is Connecting>](#).

Another thing we need to consider is the reconnect may not connect the same AP if there are more than one APs with the same SSID. The reconnect always select current best APs to connect.

5.33.13 Wi-Fi Beacon Timeout

The beacon timeout mechanism is used by ESP32 station to detect whether the AP is alive or not. If the station continuously loses 60 beacons of the connected AP, the beacon timeout happens.

After the beacon timeout happens, the station sends 5 probe requests to AP, it disconnects the AP and raises the event [<WIFI_EVENT_STA_DISCONNECTED>](#) if still no probe response or beacon is received from AP.

5.33.14 ESP32 Wi-Fi Configuration

All configurations will be stored into flash when the Wi-Fi NVS is enabled; otherwise, refer to [<Wi-Fi NVS Flash>](#).

Wi-Fi Mode

Call `esp_wifi_set_mode()` to set the Wi-Fi mode.

| Mode | Description |
|------------------|--|
| WIFI_MODE_NULL | Idle: in this mode, the internal data struct is not allocated to the station and the AP, while both the station and AP interfaces are not initialized for RX/TX Wi-Fi data. Generally, this mode is used for Sniffer, or when you only want to stop both the STA and the AP without calling <code>esp_wifi_deinit()</code> to unload the whole Wi-Fi driver. |
| WIFI_MODE_STA | Station mode: in this mode, <code>esp_wifi_start()</code> will init the internal station data, while the station's interface is ready for the RX and TX Wi-Fi data. After <code>esp_wifi_connect()</code> is called, the STA will connect to the target AP. |
| WIFI_MODE_AP | AP mode: in this mode, <code>esp_wifi_start()</code> will init the internal AP data, while the AP's interface is ready for RX/TX Wi-Fi data. Then, the Wi-Fi driver starts broadcasting beacons, and the AP is ready to get connected to other stations. |
| WIFI_MODE_AP_STA | Station-AP coexistence mode: in this mode, <code>esp_wifi_start()</code> will simultaneously init both the station and the AP. This is done in station mode and AP mode. Please note that the channel of the external AP, which the ESP32 Station is connected to, has higher priority over the ESP32 AP channel. |

Station Basic Configuration

API `esp_wifi_set_config()` can be used to configure the station. The table below describes the fields in detail.

| Field | Description |
|-------------|---|
| ssid | This is the SSID of the target AP, to which the station wants to connect to. |
| password | Password of the target AP |
| scan_method | If the scan_method is WIFI_FAST_SCAN, the scan ends when the first matched AP is found, for WIFI_ALL_CHANNEL_SCAN, the scan finds all matched APs in all channels. The default scan is WIFI_FAST_SCAN. |
| bssid_set | If bssid_set is 0, the station connects to the AP whose SSID is the same as the field “ssid”, while the field “bssid” is ignored. In all other cases, the station connects to the AP whose SSID is the same as the “ssid” field, while its BSSID is the same the “bssid” field . |
| bssid | This is valid only when bssid_set is 1; see field “bssid_set” . |
| channel | If the channel is 0, the station scans the channel 1~N to search for the target AP; otherwise, the station starts by scanning the channel whose value is the same as that of the “channel” field, and then scans others to find the target AP. If you do not know which channel the target AP is running on, set it to 0. |
| sort_method | This field is only for WIFI_ALL_CHANNEL_SCAN If the sort_method is WIFI_CONNECT_AP_BY_SIGNAL, all matched APs are sorted by signal, for AP with best signal will be connected firstly. E.g. if the station want to connect AP whose ssid is “apxx”, the scan finds two AP whose ssid equals to “apxx”, the first AP’s signal is -90dBm, the second AP’s signal is -30dBm, the station connects the second AP firstly, it doesn’t connect the first one unless it fails to connect the second one. If the sort_method is WIFI_CONNECT_AP_BY_SECURITY, all matched APs are sorted by security. E.g. if the station wants to connect AP whose ssid is “apxx”, the scan finds two AP whose ssid is “apxx”, the security of the first found AP is open while the second one is WPA2, the stations connects to the second AP firstly, it doesn’t connect the second one unless it fails to connect the first one. |
| threshold | The threshold is used to filter the found AP, if the RSSI or security mode is less than the configured threshold, the AP will be discard. If the RSSI set to 0, it means default threshold, the default RSSI threshold is -127dBm. If the authmode threshold is set to 0, it means default threshold, the default authmode threshold is open. |

Attention: WEP/WPA security modes are deprecated in IEEE802.11-2016 specifications and are recommended not to be used. These modes can be rejected using authmode threshold by setting threshold as WPA2 by threshold.authmode as WIFI_AUTH_WPA2_PSK.

AP Basic Configuration

API `esp_wifi_set_config()` can be used to configure the AP. The table below describes the fields in detail.

| Field | Description |
|-----------------|--|
| ssid | SSID of AP; if the ssid[0] is 0xFF and ssid[1] is 0xFF, the AP defaults the SSID to ESP_aabcc, where “aabcc” is the last three bytes of the AP MAC. |
| password | Password of AP; if the auth mode is WIFI_AUTH_OPEN, this field will be ignored. |
| ssid_len | Length of SSID; if ssid_len is 0, check the SSID until there is a termination character. If ssid_len > 32, change it to 32; otherwise, set the SSID length according to ssid_len. |
| channel | Channel of AP; if the channel is out of range, the Wi-Fi driver defaults the channel to channel 1. So, please make sure the channel is within the required range. For more details, refer to < Wi-Fi Country Code >. |
| auth-mode | Auth mode of ESP32 AP; currently, ESP32 Wi-Fi does not support AUTH_WEP. If the authmode is an invalid value, AP defaults the value to WIFI_AUTH_OPEN. |
| ssid_hidden | If ssid_hidden is 1, AP does not broadcast the SSID; otherwise, it does broadcast the SSID. |
| max_connection | Currently, ESP32 Wi-Fi supports up to 10 Wi-Fi connections. If max_connection > 10, AP defaults the value to 10. |
| beacon_interval | Beacon interval; the value is 100 ~ 60000 ms, with default value being 100 ms. If the value is out of range, AP defaults it to 100 ms. |

Wi-Fi Protocol Mode

Currently, the IDF supports the following protocol modes:

| Protocol Mode | Description |
|---------------|---|
| 802.11B | Call <code>esp_wifi_set_protocol(ifx, WIFI_PROTOCOL_11B)</code> to set the station/AP to 802.11B-only mode. |
| 802.11BG | Call <code>esp_wifi_set_protocol(ifx, WIFI_PROTOCOL_11B WIFI_PROTOCOL_11G)</code> to set the station/AP to 802.11BG mode. |
| 802.11BGN | Call <code>esp_wifi_set_protocol(ifx, WIFI_PROTOCOL_11B WIFI_PROTOCOL_11G WIFI_PROTOCOL_11N)</code> to set the station/AP to BGN mode. |
| 802.11BGNLR | Call <code>esp_wifi_set_protocol(ifx, WIFI_PROTOCOL_11B WIFI_PROTOCOL_11G WIFI_PROTOCOL_11N WIFI_PROTOCOL_LR)</code> to set the station/AP to BGN and the Espressif-specific mode. |
| 802.11LR | Call <code>esp_wifi_set_protocol(ifx, WIFI_PROTOCOL_LR)</code> to set the station/AP only to the Espressif-specific mode. This mode is an Espressif-patented mode which can achieve a one-kilometer line of sight range. Please, make sure both the station and the AP are connected to an ESP32 device |

Long Range (LR)

Long Range (LR) mode is an Espressif-patented Wi-Fi mode which can achieve a one-kilometer line of sight range. It has better reception sensitivity, stronger anti-interference ability and longer transmission distance than the traditional 802.11B mode.

LR Compatibility Since LR is Espressif unique Wi-Fi mode, only ESP32 devices can transmit and receive the LR data. In other words, the ESP32 device should NOT transmit the data in LR data rate if the connected device doesn't support LR. The application can achieve this by configuring suitable Wi-Fi mode. If the negotiated mode supports LR, the ESP32 may transmit data in LR rate, otherwise, ESP32 will transmit all data in traditional Wi-Fi data rate.

Following table depicts the Wi-Fi mode negotiation:

| AP\STA | BGN | BG | B | BGNLR | BGLR | BLR | LR |
|--------|-----|----|---|-------|------|-----|----|
| BGN | BGN | BG | B | BGN | BG | B | • |
| BG | BG | BG | B | BG | BG | B | • |
| B | B | B | B | B | B | B | • |
| BGNLR | • | • | • | BGNLR | BGLR | BLR | LR |
| BGLR | • | • | • | BGLR | BGLR | BLR | LR |
| BLR | • | • | • | BLR | BLR | BLR | LR |
| LR | • | • | • | LR | LR | LR | LR |

In above table, the row is the Wi-Fi mode of AP and the column is the Wi-Fi mode of station. The “-” indicates Wi-Fi mode of the AP and station are not compatible.

According to the table, we can conclude that:

- For LR enabled in ESP32 AP, it's incompatible with traditional 802.11 mode because the beacon is sent in LR mode.
- For LR enabled in ESP32 station and the mode is NOT LR only mode, it's compatible with traditional 802.11 mode.
- If both station and AP are ESP32 devices and both of them enable LR mode, the negotiated mode supports LR.

If the negotiated Wi-Fi mode supports both traditional 802.11 mode and LR mode, it's the WiFi driver's responsibility to automatically select the best data rate in different Wi-Fi mode and the application doesn't need to care about it.

LR Impacts to Traditional Wi-Fi device

The data transmission in LR rate has no impacts on the traditional Wi-Fi device because:

- The CCA and backoff process in LR mode are consistent with 802.11 specification.
- The traditional Wi-Fi device can detect the LR signal via CCA and do backoff.

In other words, the impact transmission in LR mode is similar as the impact in 802.11B mode.

LR Transmission Distance The reception sensitivity of LR has about 4 dB gain than the traditional 802.11 B mode, theoretically the transmission distance is about 2 to 2.5 times the distance of 11B.

LR Throughput The LR rate has very limited throughput because the raw PHY data rate LR is 1/2 Mbits and 1/4 Mbits.

When to Use LR

The general conditions for using LR are:

- Both the AP and station are ESP32 devices.
- Long distance WiFi connection and data transmission is required.
- Data throughput requirements are very small, such as remote device control, etc.

Wi-Fi Country Code

Call `esp_wifi_set_country()` to set the country info. The table below describes the fields in detail, please consult local 2.4GHz RF operating regulations before configuring these fields.

| Field | Description |
|--------|---|
| cc[3] | <p>Country code string, this attributes identify the country or noncountry entity in which the station/AP is operating. If it's a country, the first two octets of this string is the two character country info as described in document ISO/IEC3166-1. The third octect is one of the following:</p> <ul style="list-style-type: none"> • an ASCII space character, if the regulations under which the station/AP is operating encompass all environments for the current frequency band in the country • an ASCII 'O' character if the regulations under which the station/AP is operating are for an outdoor environment only, or • an ASCII 'I' character if the regulations under which the station/AP is operating are for an indoor environment only. • an ASCII 'X' character if the station/AP is operating under a noncountry entity. The first two octets of the noncountry entity is two ASSCII 'XX' characters. • the binary representation of the Operating Class table number currently in use. Refer 802.11-2012 Annex E. |
| schan | Start channel, it's the minimum channel number of the regulations under which the station/AP can operate. |
| snum | Total channel number of the regulations, e.g. if the schan=1, nchan=13, it means the station/AP can send data from channel 1 to 13. |
| policy | Country policy, this field control which country info will be used if the configured country info is conflict with the connected AP's. More description about policy is provided in following section. |

The default country info is {.cc=" CN" , .schan=1, .nchan=13, policy=WIFI_COUNTRY_POLICY_AUTO}, if the WiFi Mode is station/AP coexist mode, they share the same configured country info. Sometimes, the country info of AP, to which the station is connected, is different from the country info of configured. For example, the configured station has country info {.cc=" JP" , .schan=1, .nchan=14, policy=WIFI_COUNTRY_POLICY_AUTO}, but the connected AP has country info {.cc=" CN" , .schan=1, .nchan=13}, then country info of connected AP's is used. Following table depicts which country info is used in different WiFi Mode and different country policy, also describe the impact to active scan.

| WiFi Mode | Policy | Description |
|-------------------|----------------------------|---|
| Station | WIFI_COUNTRY_POLICY_AUTO | <p>If the connected AP has country IE in its beacon, the country info equals to the country info in beacon, otherwise, use default country info.</p> <p>For scan:</p> <ul style="list-style-type: none"> before the station connects to the AP, scans channel “schan” to “min(11, schan+nchan-1)” with active scan and channel min(12, schan+nchan)” to 14 with passive scan. E.g. if the used country info is {.cc=” CN” , .schan=1, .nchan=6} then 1 to 6 is active scan and 7 to 14 is passive scan If the used country info is {.cc=” CN” , .schan=1, .nchan=12} then 1 to 11 is active scan and 12 to 14 is passive scan after the station connects to the AP, scans channel “schan” to “schan+nchan-1” with active scan and channel “schan+nchan” to 14 with passive scan <p>Always keep in mind that if a AP with hidden SSID is set to a passive scan channel, the passive scan will not find it. In other words, if the application hopes to find the AP with hidden SSID in every channel, the policy of country info should be configured to WIFI_COUNTRY_POLICY_MANUAL.</p> |
| Station | WIFI_COUNTRY_POLICY_MANUAL | <p>Always use the configured country info</p> <p>For scan, scans channel “schan” to “schan+nchan-1” with active scan</p> |
| AP | WIFI_COUNTRY_POLICY_AUTO | Always use the configured country info |
| AP | WIFI_COUNTRY_POLICY_MANUAL | Always use the configured country info |
| Station/AP coexit | WIFI_COUNTRY_POLICY_AUTO | <p>If the station doesn't connect to any AP, the AP use the configured country info. If the station connects to an AP, the AP has the same country info as the station.</p> <p>Same as station mode with policy WIFI_COUNTRY_POLICY_AUTO</p> |

Home Channel In AP mode, the home channel is defined as that of the AP channel. In Station mode, the home channel is defined as the channel of the AP to which the station is connected. In Station+AP mode, the home channel of AP and station must be the same. If the home channels of Station and AP are different, the station's home channel is always in priority. Take the following as an example: at the beginning, the AP is on channel 6, then the

station connects to an AP whose channel is 9. Since the station's home channel has a higher priority, the AP needs to switch its channel from 6 to 9 to make sure that both station and AP have the same home channel. While switching channel, the ESP32 in SoftAP mode will notify the connected stations about the channel migration using a Channel Switch Announcement (CSA). Stations that support channel switching will transition smoothly whereas stations who do not will disconnect and reconnect to the SoftAP.

Wi-Fi Vendor IE Configuration

By default, all Wi-Fi management frames are processed by the Wi-Fi driver, and the application does not need to care about them. Some applications, however, may have to handle the beacon, probe request, probe response and other management frames. For example, if you insert some vendor-specific IE into the management frames, it is only the management frames which contain this vendor-specific IE that will be processed. In ESP32, `esp_wifi_set_vendor_ie()` and `esp_wifi_set_vendor_ie_cb()` are responsible for this kind of tasks.

5.33.15 Wi-Fi Security

In addition to traditional security methods (WEP/WPA-TKIP/WPA2-CCMP), ESP32 Wi-Fi now supports state-of-the-art security protocols, namely Protected Management Frames based on 802.11w standard and Wi-Fi Protected Access 3 (WPA3-Personal). Together, PMF and WPA3 provide better privacy and robustness against known attacks in traditional modes.

Protected Management Frames (PMF)

In Wi-Fi, management frames such as beacons, probes, (de)authentication, (dis)association are used by non-AP stations to scan and connect to an AP. Unlike data frames, these frames are sent unencrypted. An attacker can use eavesdropping and packet injection to send spoofed (de)authentication/(dis)association frames at the right time, leading to following attacks in case of unprotected management frame exchanges.

- DOS attack on one or all clients in the range of the attacker.
- Tearing down existing association on AP side by sending association request.
- Forcing a client to perform 4-way handshake again in case PSK is compromised in order to get PTK.
- Getting SSID of hidden network from association request.
- Launching man-in-the-middle attack by forcing clients to deauth from legitimate AP and associating to a rogue one.

PMF provides protection against these attacks by encrypting unicast management frames and providing integrity checks for broadcast management frames. These include deauthentication, disassociation and robust management frames. It also provides Secure Association (SA) teardown mechanism to prevent spoofed association/authentication frames from disconnecting already connected clients.

ESP32 supports the following three modes of operation with respect to PMF.

- PMF not supported: In this mode, ESP32 indicates to AP that it is not capable of supporting management protection during association. In effect, security in this mode will be equivalent to that in traditional mode.
- PMF capable, but not required: In this mode, ESP32 indicates to AP that it is capable of supporting PMF. The management protection will be used if AP mandates PMF or is at least capable of supporting PMF.
- PMF capable and required: In this mode, ESP32 will only connect to AP, if AP supports PMF. If not, ESP32 will refuse to connect to the AP.

`esp_wifi_set_config()` can be used to configure PMF mode by setting appropriate flags in `pmf_cfg` parameter. Currently, PMF is supported only in Station mode.

WPA3-Personal

Wi-Fi Protected Access-3 (WPA3) is a set of enhancements to Wi-Fi access security intended to replace the current WPA2 standard. In order to provide more robust authentication, WPA3 uses Simultaneous Authentication of

Equals (SAE), which is password-authenticated key agreement method based on Diffie-Hellman key exchange. Unlike WPA2, the technology is resistant to offline-dictionary attack, where the attacker attempts to determine shared password based on captured 4-way handshake without any further network interaction. WPA3 also provides forward secrecy, which means the captured data cannot be decrypted even if password is compromised after data transmission. Please refer to [Security](#) section of Wi-Fi Alliance's official website for further details.

In order to enable WPA3-Personal, "Enable WPA3-Personal" should be selected in menuconfig. If enabled, ESP32 uses SAE for authentication if supported by the AP. Since PMF is a mandatory requirement for WPA3, PMF capability should be at least set to "PMF capable, but not required" for ESP32 to use WPA3 mode. Application developers need not worry about the underlying security mode as highest available is chosen from security standpoint. Note that Wi-Fi stack size requirement will increase approximately by 3k when WPA3 is used. Currently, WPA3 is supported only in Station mode.

5.33.16 ESP32 Wi-Fi Power-saving Mode

Station Sleep

Currently, ESP32 Wi-Fi supports the Modem-sleep mode which refers to the legacy power-saving mode in the IEEE 802.11 protocol. Modem-sleep mode works in Station-only mode and the station must connect to the AP first. If the Modem-sleep mode is enabled, station will switch between active and sleep state periodically. In sleep state, RF, PHY and BB are turned off in order to reduce power consumption. Station can keep connection with AP in modem-sleep mode.

Modem-sleep mode includes minimum and maximum power save modes. In minimum power save mode, station wakes up every DTIM to receive beacon. Broadcast data will not be lost because it is transmitted after DTIM. However, it can not save much more power if DTIM is short for DTIM is determined by AP.

In maximum power save mode, station wakes up every listen interval to receive beacon. This listen interval can be set longer than the AP DTIM period. Broadcast data may be lost because station may be in sleep state at DTIM time. If listen interval is longer, more power is saved but broadcast data is more easy to lose. Listen interval can be configured by calling API `esp_wifi_set_config()` before connecting to AP.

Call `esp_wifi_set_ps(WIFI_PS_MIN_MODEM)` to enable Modem-sleep minimum power save mode or `esp_wifi_set_ps(WIFI_PS_MAX_MODEM)` to enable Modem-sleep maximum power save mode after calling `esp_wifi_init()`. When station connects to AP, Modem-sleep will start. When station disconnects from AP, Modem-sleep will stop.

Call `esp_wifi_set_ps(WIFI_PS_NONE)` to disable modem sleep entirely. This has much higher power consumption, but provides minimum latency for receiving Wi-Fi data in real time. When modem sleep is enabled, received Wi-Fi data can be delayed for as long as the DTIM period (minimum power save mode) or the listen interval (maximum power save mode). Disabling modem sleep entirely is not possible for Wi-Fi and Bluetooth coexist mode.

The default Modem-sleep mode is `WIFI_PS_MIN_MODEM`.

AP Sleep

Currently ESP32 AP doesn't support all of the power save feature defined in Wi-Fi specification. To be specific, the AP only caches unicast data for the stations connect to this AP, but doesn't cache the multicast data for the stations. If stations connected to the ESP32 AP are power save enabled, they may experience multicast packet loss.

In future, all power save features will be supported on ESP32 AP.

5.33.17 ESP32 Wi-Fi Connect Crypto

Now ESP32 have two group crypto functions can be used when do wifi connect, one is the original functions, the other is optimized by ESP hardware: 1. Original functions which is the source code used in the folder `components/wpa_supplicant/src/crypto` function; 2. The optimized functions is in the folder `components/wpa_supplicant/src/fast_crypto`, these function used the hardware crypto to make it faster than origin one,

the type of function's name add *fast_* to distinguish with the original one. For example, the API `aes_wrap()` is used to encrypt frame information when do 4 way handshake, the `fast_aes_wrap()` has the same result but can be faster.

Two groups of crypto function can be used when register in the `wpa_crypto_funcs_t`, `wpa2_crypto_funcs_t` and `wps_crypto_funcs_t` structure, also we have given the recommend functions to register in the `fast_crypto_ops.c`, you can register the function as the way you need, however what should make action is that the `crypto_hash_xxx` function and `crypto_cipher_xxx` function need to register with the same function to operation. For example, if you register `crypto_hash_init()` function to initialize the `esp_crypto_hash` structure, you need use the `crypto_hash_update()` and `crypto_hash_finish()` function to finish the operation, rather than `fast_crypto_hash_update()` or `fast_crypto_hash_finish()`.

5.33.18 ESP32 Wi-Fi Throughput

The table below shows the best throughput results we got in Espressif's lab and in a shield box.

| Type/Throughput | Air In Lab | Shield-box | Test Tool | IDF Version (commit ID) |
|----------------------|-------------|---------------------|---------------|-------------------------|
| Raw 802.11 Packet RX | N/A | 130 MBit/sec | Internal tool | NA |
| Raw 802.11 Packet TX | N/A | 130 MBit/sec | Internal tool | NA |
| UDP RX | 30 MBit/sec | 90 MBit/sec | iperf example | 05838641 |
| UDP TX | 30 MBit/sec | 60 MBit/sec | iperf example | 05838641 |
| TCP RX | 20 MBit/sec | 50 MBit/sec | iperf example | 05838641 |
| TCP TX | 20 MBit/sec | 50 MBit/sec | iperf example | 05838641 |

When the throughput is tested by iperf example, the `sdkconfig` is [examples/wifi/iperf/sdkconfig.defaults.99](#)

5.33.19 Wi-Fi 80211 Packet Send

The `esp_wifi_80211_tx()` API can be used to:

- Send the beacon, probe request, probe response, action frame.
- Send the non-QoS data frame.

It cannot be used for sending encrypted or QoS frames.

Preconditions of Using `esp_wifi_80211_tx`

- The Wi-Fi mode is Station, or AP, or Station+AP.
- Either `esp_wifi_set_promiscuous(true)`, or `esp_wifi_start()`, or both of these APIs return `ESP_OK`. This is because we need to make sure that Wi-Fi hardware is initialized before `esp_wifi_80211_tx()` is called. In ESP32, both `esp_wifi_set_promiscuous(true)` and `esp_wifi_start()` can trigger the initialization of Wi-Fi hardware.
- The parameters of `esp_wifi_80211_tx` are hereby correctly provided.

Data rate

- If there is no WiFi connection, the data rate is 1Mbps.
- If there is WiFi connection and the packet is from station to AP or from AP to station, the data rate is same as the WiFi connection. Otherwise the data rate is 1Mbps.

Side-Effects to Avoid in Different Scenarios

Theoretically, if we do not consider the side-effects the API imposes on the Wi-Fi driver or other stations/APs, we can send a raw 802.11 packet over the air, with any destination MAC, any source MAC, any BSSID, or any other type of packet. However, robust/useful applications should avoid such side-effects. The table below provides some tips/recommendations on how to avoid the side-effects of `esp_wifi_80211_tx` in different scenarios.

| Scenario | Description |
|----------------------|--|
| No WiFi connection | <p>In this scenario, no Wi-Fi connection is set up, so there are no side-effects on the Wi-Fi driver. If <code>en_sys_seq==true</code>, the Wi-Fi driver is responsible for the sequence control. If <code>en_sys_seq==false</code>, the application needs to ensure that the buffer has the correct sequence.</p> <p>Theoretically, the MAC address can be any address. However, this may impact other stations/APs with the same MAC/BSSID.</p> <p>Side-effect example#1 The application calls <code>esp_wifi_80211_tx</code> to send a beacon with BSSID == <code>mac_x</code> in AP mode, but the <code>mac_x</code> is not the MAC of the AP interface. Moreover, there is another AP, say “other-AP”, whose bssid is <code>mac_x</code>. If this happens, an “unexpected behavior” may occur, because the stations which connect to the “other-AP” cannot figure out whether the beacon is from the “other-AP” or the <code>esp_wifi_80211_tx</code>.</p> <p>To avoid the above-mentioned side-effects, we recommend that:</p> <ul style="list-style-type: none"> • If <code>esp_wifi_80211_tx</code> is called in Station mode, the first MAC should be a multicast MAC or the exact target-device’s MAC, while the second MAC should be that of the station interface. • If <code>esp_wifi_80211_tx</code> is called in AP mode, the first MAC should be a multicast MAC or the exact target-device’s MAC, while the second MAC should be that of the AP interface. <p>The recommendations above are only for avoiding side-effects and can be ignored when there are good reasons for doing this.</p> |
| Have WiFi connection | <p>When the Wi-Fi connection is already set up, and the sequence is controlled by the application, the latter may impact the sequence control of the Wi-Fi connection, as a whole. So, the <code>en_sys_seq</code> need to be true, otherwise <code>ESP_ERR_WIFI_ARG</code> is returned.</p> <p>The MAC-address recommendations in the “No WiFi connection” scenario also apply to this scenario.</p> <p>If the WiFi mode is station mode and the MAC address1 is the MAC of AP to which the station is connected, the MAC address2 is the MAC of station interface, we say the packets is from the station to AP. On the other hand, if the WiFi mode is AP mode and the MAC address1 is the MAC of the station who connects to this AP, the MAC address2 is the MAC of AP interface, we say the packet is from the AP to station. To avoid conflicting with WiFi connections, the following checks are applied:</p> <ul style="list-style-type: none"> • If the packet type is data and is from the station to AP, the ToDS bit in ieee80211 frame control should be 1, the FromDS bit should be 0, otherwise the packet will be discarded by WiFi driver. • If the packet type is data and is from the AP to station, the ToDS bit in ieee80211 frame control should be 0, the FromDS bit should be 1, otherwise the packet will be discarded by WiFi driver. • If the packet is from station to AP or from AP to station, the Power Management, More Data, |
| Espressif Systems | <p>1717 Re-Transmission bits should be 0, otherwise the packet will be discarded by WiFi driver. ESP_ERR_WIFI_ARG is returned if any check fails.</p> |

5.33.20 Wi-Fi Sniffer Mode

The Wi-Fi sniffer mode can be enabled by `esp_wifi_set_promiscuous()`. If the sniffer mode is enabled, the following packets **can** be dumped to the application:

- 802.11 Management frame
- 802.11 Data frame, including MPDU, AMPDU, AMSDU, etc.
- 802.11 MIMO frame, for MIMO frame, the sniffer only dumps the length of the frame.
- 802.11 Control frame

The following packets will **NOT** be dumped to the application:

- 802.11 error frame, such as the frame with a CRC error, etc.

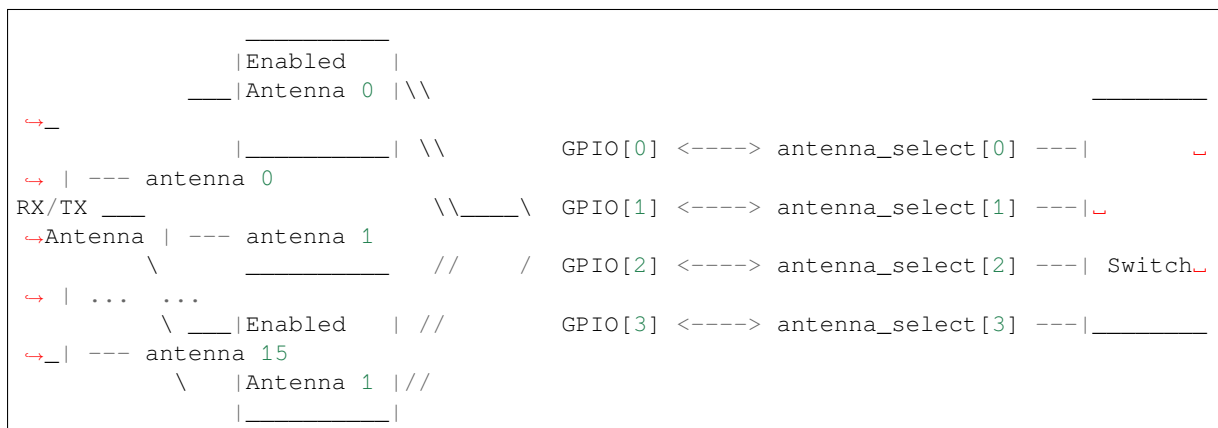
For frames that the sniffer **can** dump, the application can additionally decide which specific type of packets can be filtered to the application by using `esp_wifi_set_promiscuous_filter()` and `esp_wifi_set_promiscuous_ctrl_filter()`. By default, it will filter all 802.11 data and management frames to the application.

The Wi-Fi sniffer mode can be enabled in the Wi-Fi mode of `WIFI_MODE_NULL`, or `WIFI_MODE_STA`, or `WIFI_MODE_AP`, or `WIFI_MODE_APSTA`. In other words, the sniffer mode is active when the station is connected to the AP, or when the AP has a Wi-Fi connection. Please note that the sniffer has a **great impact** on the throughput of the station or AP Wi-Fi connection. Generally, we should **NOT** enable the sniffer, when the station/AP Wi-Fi connection experiences heavy traffic unless we have special reasons.

Another noteworthy issue about the sniffer is the callback `wifi_promiscuous_cb_t`. The callback will be called directly in the Wi-Fi driver task, so if the application has a lot of work to do for each filtered packet, the recommendation is to post an event to the application task in the callback and defer the real work to the application task.

5.33.21 Wi-Fi Multiple Antennas

The Wi-Fi multiple antennas selecting can be depicted as following picture:



ESP32 supports up to sixteen antennas through external antenna switch. The antenna switch can be controlled by up to four address pins - `antenna_select[0:3]`. Different input value of `antenna_select[0:3]` means selecting different antenna. E.g. the value '0b1011' means the antenna 11 is selected. The default value of `antenna_select[3:0]` is '0b0000', it means the antenna 0 is selected by default.

Up to four GPIOs are connected to the four active high `antenna_select` pins. ESP32 can select the antenna by control the GPIO[0:3]. The API `esp_wifi_set_ant_gpio()` is used to configure which GPIOs are connected to `antenna_selects`. If `GPIO[x]` is connected to `antenna_select[x]`, then `gpio_config->gpio_cfg[x].gpio_select` should be set to 1 and `gpio_config->gpio_cfg[x].gpio_num` should be provided.

Although up to sixteen antennas are supported, only one or two antennas can be simultaneously enabled for RX/TX. The API `esp_wifi_set_ant()` is used to configure which antennas are enabled.

The enabled antennas selecting algorithm is also configured by `esp_wifi_set_ant()`. The RX/TX antenna mode can be `WIFI_ANT_MODE_ANT0`, `WIFI_ANT_MODE_ANT1` or `WIFI_ANT_MODE_AUTO`. If the antenna mode is `WIFI_ANT_MODE_ANT0`, the enabled antenna 0 is selected for RX/TX data. If the antenna mode

is `WIFI_ANT_MODE_ANT1`, the enabled antenna 1 is selected for RX/TX data. Otherwise, WiFi automatically selects the antenna that has better signal from the enabled antennas.

If the RX antenna mode is `WIFI_ANT_MODE_AUTO`, the default antenna mode also needs to be set. Because the RX antenna switching only happens when some conditions are met, e.g. the RX antenna starts to switch if the RSSI is lower than -65dBm and if another antenna has better signal etc, RX uses the default antenna if the conditions are not met. If the default antenna mode is `WIFI_ANT_MODE_ANT1`, the enabled antenna 1 is used as the default RX antenna, otherwise the enabled antenna 0 is used as the default RX antenna.

Some limitations need to be considered:

- The TX antenna can be set to `WIFI_ANT_MODE_AUTO` only if the RX antenna mode is `WIFI_ANT_MODE_AUTO` because TX antenna selecting algorithm is based on RX antenna in `WIFI_ANT_MODE_AUTO` type.
- Currently BT doesn't support the multiple antennas feature, please don't use multiple antennas related APIs.

Following is the recommended scenarios to use the multiple antennas:

- In Wi-Fi mode `WIFI_MODE_STA`, both RX/TX antenna modes are configured to `WIFI_ANT_MODE_AUTO`. The WiFi driver selects the better RX/TX antenna automatically.
- The RX antenna mode is configured to `WIFI_ANT_MODE_AUTO`. The TX antenna mode is configured to `WIFI_ANT_MODE_ANT0` or `WIFI_ANT_MODE_ANT1`. The applications can choose to always select a specified antenna for TX, or implement their own TX antenna selecting algorithm, e.g. selecting the TX antenna mode based on the channel switch information etc.
- Both RX/TX antenna modes are configured to `WIFI_ANT_MODE_ANT0` or `WIFI_ANT_MODE_ANT1`.

Wi-Fi Multiple Antennas Configuration

Generally, following steps can be taken to configure the multiple antennas:

- Configure which GPIOs are connected to the antenna_selects, for example, if four antennas are supported and GPIO20/GPIO21 are connected to antenna_select[0]/antenna_select[1], the configurations look like:

```
wifi_ant_gpio_config_t config = {
    { .gpio_select = 1, .gpio_num = 20 },
    { .gpio_select = 1, .gpio_num = 21 }
};
```

- Configure which antennas are enabled and how RX/TX use the enabled antennas, for example, if antenna1 and antenna3 are enabled, the RX needs to select the better antenna automatically and uses antenna1 as its default antenna, the TX always selects the antenna3. The configuration looks like:

```
wifi_ant_config_t config = {
    .rx_ant_mode = WIFI_ANT_MODE_AUTO,
    .rx_ant_default = WIFI_ANT_ANT0,
    .tx_ant_mode = WIFI_ANT_MODE_ANT1,
    .enabled_ant0 = 1,
    .enabled_ant1 = 3
};
```

5.33.22 Wi-Fi Channel State Information

Channel state information (CSI) refers to the channel information of a Wi-Fi connection. In ESP32, this information consists of channel frequency responses of sub-carriers and is estimated when packets are received from the transmitter. Each channel frequency response of sub-carrier is recorded by two bytes of signed characters. The first one is imaginary part and the second one is real part. There are up to three fields of channel frequency responses according to the type of received packet. They are legacy long training field (LLTF), high throughput LTF (HT-LTF) and space time block code HT-LTF (STBC-HT-LTF). For different types of packets which are received on channels with different state, the sub-carrier index and total bytes of signed characters of CSI is shown in the following table.

| channel information | secondary channel mode | none | | | below | | | | | above | | | | |
|---------------------------|------------------------------|-------------------|-------------------|-------------------|-------------|-------------|-------|-------------------|-------------------|-------------|-----------|-------------|-------------------|-------------------|
| | | non HT | HT | | non HT | HT | | non HT | HT | | non HT | HT | | |
| channel band- width | STBC | 20MHz | 20MHz | 20MHz | 20MHz | 20MHz | 40MHz | 20MHz | 20MHz | 40MHz | 20MHz | 20MHz | 40MHz | |
| | | non STBC | non STBC | STBC | non STBC | non STBC | STBC | non STBC | STBC | non STBC | STBC | non STBC | non STBC | STBC |
| sub- carrier index | LLTF | 0~31, 32~ 1 | 0~31, 32~ 1 | 0~31, 32~ 1 | 0~63 | 0~63 | 0~63 | 0~63 | 0~63 | - | - | - | - | - |
| | HT- LTF | • | 0~31, 32~ 1 | 0~31, 32~ 1 | • | 0~63 | 0~62 | 0~63, 64~ 1 | 0~60, 60~ 1 | • | - | - | 0~63, 64~ 1 | 0~60, 60~ 1 |
| | STBC- HT- LTF | • | • | 0~31, 32~ 1 | • | • | 0~62 | • | 0~60, 60~ 1 | • | • | - | 62~ 1 | • |
| total bytes | | 128 | 256 | 384 | 128 | 256 | 380 | 384 | 612 | 128 | 256 | 376 | 384 | 612 |

All of the information in the table can be found in the structure `wifi_csi_info_t`.

- Secondary channel refers to `secondary_channel` field of `rx_ctrl` field.
- Signal mode of packet refers to `sig_mode` field of `rx_ctrl` field.
- Channel bandwidth refers to `cwb` field of `rx_ctrl` field.
- STBC refers to `stbc` field of `rx_ctrl` field.
- Total bytes refers to `len` field.
- The CSI data corresponding to each Long Training Field(LTF) type is stored in a buffer starting from the `buf` field. Each item is stored as two bytes: imaginary part followed by real part. The order of each item is the same as the sub-carrier in the table. The order of LTF is: LLTF, HT-LTF, STBC-HT-LTF. However all 3 LTFs may not be present, depending on the channel and packet information (see above).
- If `first_word_invalid` field of `wifi_csi_info_t` is true, it means that the first four bytes of CSI data is invalid due to a hardware limitation in ESP32.
- More information like RSSI, noise floor of RF, receiving time and antenna is in the `rx_ctrl` field.

Note:

- For STBC packet, CSI is provided for every space-time stream without CSD (cyclic shift delay). As each cyclic shift on the additional chains shall be -200ns, only the CSD angle of first space-time stream is recorded in sub-carrier 0 of HT-LTF and STBC-HT-LTF for there is no channel frequency response in sub-carrier 0. CSD[10:0] is 11 bits, ranging from $-\pi$ to π .
- If LLTF, HT-LTF or STBC-HT-LTF is not enabled by calling API `esp_wifi_set_csi_config()`, the total bytes of CSI data will be fewer than that in the table. For example, if LLTF and HT-LTF is not enabled and STBC-HT-LTF is enabled, when a packet is received with the condition above/HT/40MHz/STBC, the total bytes of CSI data is 244 $((61 + 60) * 2 + 2 = 244$, the result is aligned to four bytes and the last two bytes is invalid).

5.33.23 Wi-Fi Channel State Information Configure

To use Wi-Fi CSI, the following steps need to be done.

- Select Wi-Fi CSI in menuconfig. It is “Menuconfig → Components config → Wi-Fi → WiFi CSI(Channel State Information)” .
- Set CSI receiving callback function by calling API `esp_wifi_set_csi_rx_cb()`.

- Configure CSI by calling API `esp_wifi_set_csi_config()`.
- Enable CSI by calling API `esp_wifi_set_csi()`.

The CSI receiving callback function runs from Wi-Fi task. So, do not do lengthy operations in the callback function. Instead, post necessary data to a queue and handle it from a lower priority task. Because station does not receive any packet when it is disconnected and only receives packets from AP when it is connected, it is suggested to enable sniffer mode to receive more CSI data by calling `esp_wifi_set_promiscuous()`.

5.33.24 Wi-Fi HT20/40

ESP32 supports Wi-Fi bandwidth HT20 or HT40, it doesn't support HT20/40 coexist. `esp_wifi_set_bandwidth` can be used to change the default bandwidth of station or AP. The default bandwidth for ESP32 station and AP is HT40.

In station mode, the actual bandwidth is firstly negotiated during the Wi-Fi connection. It is HT40 only if both the station and the connected AP support HT40, otherwise it's HT20. If the bandwidth of connected AP is changes, the actual bandwidth is negotiated again without Wi-Fi disconnecting.

Similarly, in AP mode, the actual bandwidth is negotiated between AP and the stations that connect to the AP. It's HT40 if the AP and one of the stations support HT40, otherwise it's HT20.

In station/AP coexist mode, the station/AP can configure HT20/40 separately. If both station and AP are negotiated to HT40, the HT40 channel should be the channel of station because the station always has higher priority than AP in ESP32. E.g. the configured bandwidth of AP is HT40, the configured primary channel is 6 and the configured secondary channel is 10. The station is connected to an router whose primary channel is 6 and secondary channel is 2, then the actual channel of AP is changed to primary 6 and secondary 2 automatically.

Theoretically the HT40 can gain better throughput because the maximum raw physical (PHY) data rate for HT40 is 150Mbps while it's 72Mbps for HT20. However, if the device is used in some special environment, e.g. there are too many other Wi-Fi devices around the ESP32 device, the performance of HT40 may be degraded. So if the applications need to support same or similar scenarios, it's recommended that the bandwidth is always configured to HT20.

5.33.25 Wi-Fi QoS

ESP32 supports all the mandatory features required in WFA Wi-Fi QoS Certification.

Four ACs(Access Category) are defined in Wi-Fi specification, each AC has a its own priority to access the Wi-Fi channel. Moreover a map rule is defined to map the QoS priority of other protocol, such as 802.11D or TCP/IP precedence to Wi-Fi AC.

Below is a table describes how the IP Precedences are mapped to Wi-Fi ACs in ESP32, it also indicates whether the AMPDU is supported for this AC. The table is sorted with priority descending order, namely, the AC_VO has highest priority.

| IP Precedence | Wi-Fi AC | Support AMPDU? |
|---------------|---------------------|----------------|
| 6, 7 | AC_VO (Voice) | No |
| 4, 5 | AC_VI (Video) | Yes |
| 3, 0 | AC_BE (Best Effort) | Yes |
| 1, 2 | AC_BK (Background) | Yes |

The application can make use of the QoS feature by configuring the IP precedence via socket option IP_TOS. Here is an example to make the socket to use VI queue:

```
const int ip_precedence_vi = 4;
const int ip_precedence_offset = 5;
int priority = (ip_precedence_vi << ip_precedence_offset);
setsockopt(socket_id, IPPROTO_IP, IP_TOS, &priority, sizeof(priority));
```

Theoretically the higher priority AC has better performance than the low priority AC, however, it's not always be true, here

- For some really important application traffic, can put it into AC_VO queue. Avoid sending big traffic via AC_VO queue. On one hand, the AC_VO queue doesn't support AMPDU and it can't get better performance than other queue if the traffic is big, on the other hand, it may impact the the management frames that also use AC_VO queue.
- Avoid using more than two different AMPDU supported precedences, e.g. socket A uses precedence 0, socket B uses precedence 1, socket C uses precedence 2, this is a bad design because it may need much more memory. To be detailed, the Wi-Fi driver may generate a Block Ack session for each precedence and it needs more memory if the Block Ack session is setup.

5.33.26 Wi-Fi AMSDU

ESP32 supports receiving AMSDU but doesn't support transmitting AMSDU. The transmitting AMSDU is not necessary since ESP32 has transmitting AMPDU.

5.33.27 Wi-Fi Fragment

ESP32 supports Wi-Fi receiving fragment, but doesn't support Wi-Fi transmitting fragment. The Wi-Fi transmitting fragment will be supported in future release.

5.33.28 WPS Enrolle

ESP32 supports WPS enrollee feature in Wi-Fi mode WIFI_MODE_STA or WIFI_MODE_APSTA. Currently ESP32 supports WPS enrollee type PBC and PIN.

5.33.29 Wi-Fi Buffer Usage

This section is only about the dynamic buffer configuration.

Why Buffer Configuration Is Important

In order to get a robust, high-performance system, we need to consider the memory usage/configuration very carefully, because

- the available memory in ESP32 is limited.
- currently, the default type of buffer in LwIP and Wi-Fi drivers is “dynamic” , **which means that both the LwIP and Wi-Fi share memory with the application**. Programmers should always keep this in mind; otherwise, they will face a memory issue, such as “running out of heap memory” .
- it is very dangerous to run out of heap memory, as this will cause ESP32 an “undefined behavior” . Thus, enough heap memory should be reserved for the application, so that it never runs out of it.
- the Wi-Fi throughput heavily depends on memory-related configurations, such as the TCP window size, Wi-Fi RX/TX dynamic buffer number, etc.
- the peak heap memory that the ESP32 LwIP/Wi-Fi may consume depends on a number of factors, such as the maximum TCP/UDP connections that the application may have, etc.
- the total memory that the application requires is also an important factor when considering memory configuration.

Due to these reasons, there is not a good-for-all application configuration. Rather, we have to consider memory configurations separately for every different application.

Dynamic vs. Static Buffer

The default type of buffer in Wi-Fi drivers is “dynamic” . Most of the time the dynamic buffer can significantly save memory. However, it makes the application programming a little more difficult, because in this case the application needs to consider memory usage in Wi-Fi.

lwIP also allocates buffers at the TCP/IP layer, and this buffer allocation is also dynamic. See [lwIP documentation section about memory use and performance](#).

Peak Wi-Fi Dynamic Buffer

The Wi-Fi driver supports several types of buffer (refer to [Wi-Fi Buffer Configure](#)). However, this section is about the usage of the dynamic Wi-Fi buffer only. The peak heap memory that Wi-Fi consumes is the **theoretically-maximum memory** that the Wi-Fi driver consumes. Generally, the peak memory depends on:

- the number of dynamic rx buffers that are configured: `wifi_rx_dynamic_buf_num`
- the number of dynamic tx buffers that are configured: `wifi_tx_dynamic_buf_num`
- the maximum packet size that the Wi-Fi driver can receive: `wifi_rx_pkt_size_max`
- the maximum packet size that the Wi-Fi driver can send: `wifi_tx_pkt_size_max`

So, the peak memory that the Wi-Fi driver consumes can be calculated with the following formula:

$$\text{wifi_dynamic_peek_memory} = (\text{wifi_rx_dynamic_buf_num} * \text{wifi_rx_pkt_size_max}) + (\text{wifi_tx_dynamic_buf_num} * \text{wifi_tx_pkt_size_max})$$

Generally, we do not need to care about the dynamic tx long buffers and dynamic tx long long buffers, because they are management frames which only have a small impact on the system.

5.33.30 Wi-Fi Menuconfig

Wi-Fi Buffer Configure

If you are going to modify the default number or type of buffer, it would be helpful to also have an overview of how the buffer is allocated/freed in the data path. The following diagram shows this process in the TX direction:

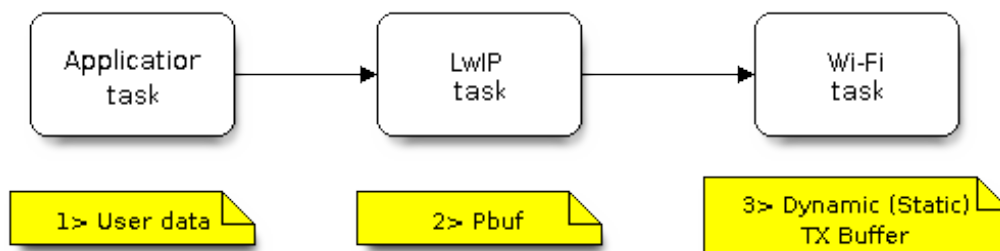


Fig. 61: TX Buffer Allocation

Description:

- The application allocates the data which needs to be sent out.
- The application calls TCP/IP-/Socket-related APIs to send the user data. These APIs will allocate a PBUF used in LwIP, and make a copy of the user data.
- When LwIP calls a Wi-Fi API to send the PBUF, the Wi-Fi API will allocate a “Dynamic Tx Buffer” or “Static Tx Buffer”, make a copy of the LwIP PBUF, and finally send the data.

The following diagram shows how buffer is allocated/freed in the RX direction:

Description:

- The Wi-Fi hardware receives a packet over the air and puts the packet content to the “Static Rx Buffer”, which is also called “RX DMA Buffer”.
- The Wi-Fi driver allocates a “Dynamic Rx Buffer”, makes a copy of the “Static Rx Buffer”, and returns the “Static Rx Buffer” to hardware.
- The Wi-Fi driver delivers the packet to the upper-layer (LwIP), and allocates a PBUF for holding the “Dynamic Rx Buffer”.
- The application receives data from LwIP.

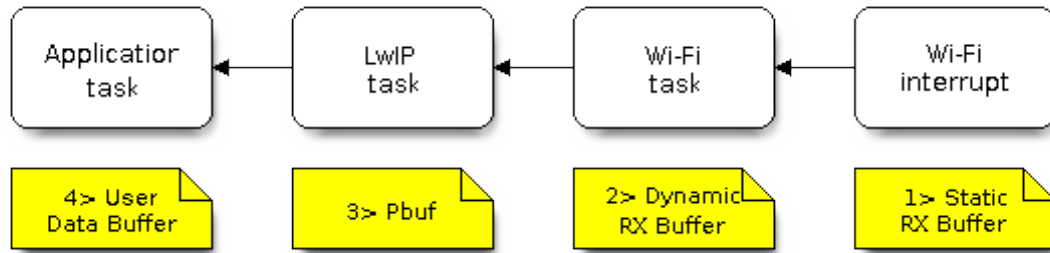


Fig. 62: RX Buffer Allocation

The diagram shows the configuration of the Wi-Fi internal buffer.

| Buffer Type | Alloc Type | Default | Configurable | Description |
|---------------------------------------|------------|-----------------|--------------|--|
| Static RX Buffer (Hardware RX Buffer) | Static | 10 * 1600 Bytes | Yes | <p>This is a kind of DMA memory. It is initialized in <code>esp_wifi_init()</code> and freed in <code>esp_wifi_deinit()</code>. The 'Static Rx Buffer' forms the hardware receiving list. Upon receiving a frame over the air, hardware writes the frame into the buffer and raises an interrupt to the CPU. Then, the Wi-Fi driver reads the content from the buffer and returns the buffer back to the list.</p> <p>If the application want to reduce the the memory statically allocated by Wi-Fi, they can reduce this value from 10 to 6 to save 6400 Bytes memory. It's not recommended to reduce the configuration to a value less than 6 unless the AMPDU feature is disabled.</p> |
| Dynamic RX Buffer | Dynamic | 32 | Yes | <p>The buffer length is variable and it depends on the received frames' length. When the Wi-Fi driver receives a frame from the 'Hardware Rx Buffer', the 'Dynamic Rx Buffer' needs to be allocated from the heap. The number of the Dynamic Rx Buffer, configured in the <code>menuconfig</code>, is used to limit the total un-freed Dynamic Rx Buffer number.</p> |
| Dynamic TX Buffer | Dynamic | 32 | Yes | <p>This is a kind of DMA memory. It is allocated to the heap. When the upper-layer (LwIP) sends packets to the Wi-Fi driver, it firstly allocates</p> |
| Espressif Systems | | 1725 | | Release v4.1.3 |

Wi-Fi NVS Flash

If the Wi-Fi NVS flash is enabled, all Wi-Fi configurations set via the Wi-Fi APIs will be stored into flash, and the Wi-Fi driver will start up with these configurations next time it powers on/reboots. However, the application can choose to disable the Wi-Fi NVS flash if it does not need to store the configurations into persistent memory, or has its own persistent storage, or simply due to debugging reasons, etc.

Wi-Fi AMPDU

ESP32 supports both receiving and transmitting AMPDU, the AMPDU can greatly improve the Wi-Fi throughput. Generally, the AMPDU should be enabled. Disabling AMPDU is usually for debugging purposes.

5.33.31 Troubleshooting

Please refer to a separate document with *Espressif Wireshark User Guide*.

Espressif Wireshark User Guide

1. Overview

1.1 What is Wireshark? **Wireshark** (originally named “Ethereal”) is a network packet analyzer that captures network packets and displays the packet data as detailed as possible. It uses WinPcap as its interface to directly capture network traffic going through a network interface controller (NIC).

You could think of a network packet analyzer as a measuring device used to examine what is going on inside a network cable, just like a voltmeter is used by an electrician to examine what is going on inside an electric cable.

In the past, such tools were either very expensive, proprietary, or both. However, with the advent of Wireshark, all that has changed.

Wireshark is released under the terms of the GNU General Public License, which means you can use the software and the source code free of charge. It also allows you to modify and customize the source code.

Wireshark is, perhaps, one of the best open source packet analyzers available today.

1.2 Some Intended Purposes Here are some examples of how Wireshark is typically used:

- Network administrators use it to troubleshoot network problems.
- Network security engineers use it to examine security problems.
- Developers use it to debug protocol implementations.
- People use it to learn more about network protocol internals.

Beside these examples, Wireshark can be used for many other purposes.

1.3 Features The main features of Wireshark are as follows:

- Available for UNIX and Windows
- Captures live packet data from a network interface
- Displays packets along with detailed protocol information
- Opens/saves the captured packet data
- Imports/exports packets into a number of file formats, supported by other capture programs
- Advanced packet filtering
- Searches for packets based on multiple criteria
- Colorizes packets according to display filters
- Calculates statistics
- ...and a lot more!

1.4 Wireshark Can or Can't Do

- **Live capture from different network media.**
Wireshark can capture traffic from different network media, including wireless LAN.
- **Import files from many other capture programs.**
Wireshark can import data from a large number of file formats, supported by other capture programs.
- **Export files for many other capture programs.**
Wireshark can export data into a large number of file formats, supported by other capture programs.
- **Numerous protocol dissectors.**
Wireshark can dissect, or decode, a large number of protocols.
- **Wireshark is not an intrusion detection system.**
It will not warn you if there are any suspicious activities on your network. However, if strange things happen, Wireshark might help you figure out what is really going on.
- **Wireshark does not manipulate processes on the network, it can only perform “measurements” within it.**
Wireshark does not send packets on the network or influence it in any other way, except for resolving names (converting numerical address values into a human readable format), but even that can be disabled.

2. Where to Get Wireshark You can get Wireshark from the official website: <https://www.wireshark.org/download.html>

Wireshark can run on various operating systems. Please download the correct version according to the operating system you are using.

3. Step-by-step Guide This demonstration uses **Wireshark 2.2.6 on Linux.**

a) Start Wireshark

On Linux, you can run the shell script provided below. It starts Wireshark, then configures NIC and the channel for packet capture.

```
ifconfig $1 down
iwconfig $1 mode monitor
iwconfig $1 channel $2
ifconfig $1 up
Wireshark&
```

In the above script, the parameter \$1 represents NIC and \$2 represents channel. For example, wlan0 in ./xxx.sh wlan0 6, specifies the NIC for packet capture, and 6 identifies the channel of an AP or Soft-AP.

b) Run the Shell Script to Open Wireshark and Display Capture Interface

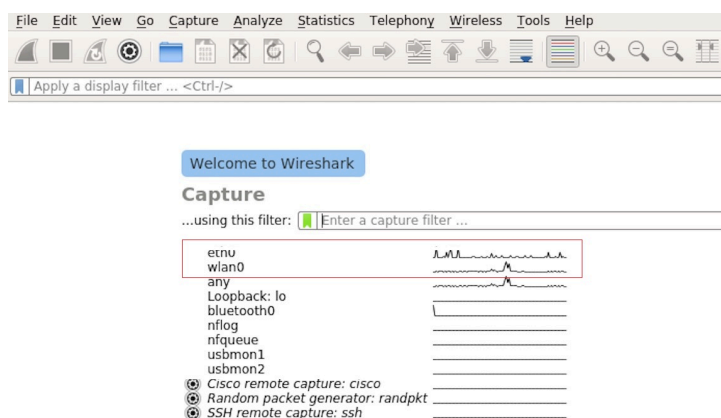


Fig. 63: Wireshark Capture Interface

c) Select the Interface to Start Packet Capture

As the red markup shows in the picture above, many interfaces are available. The first one is a local NIC and the second one is a wireless NIC.

Please select the NIC according to your requirements. This document will use the wireless NIC to demonstrate packet capture.

Double click `wlan0` to start packet capture.

d) Set up Filters

Since all packets in the channel will be captured, and many of them are not needed, you have to set up filters to get the packets that you need.

Please find the picture below with the red markup, indicating where the filters should be set up.

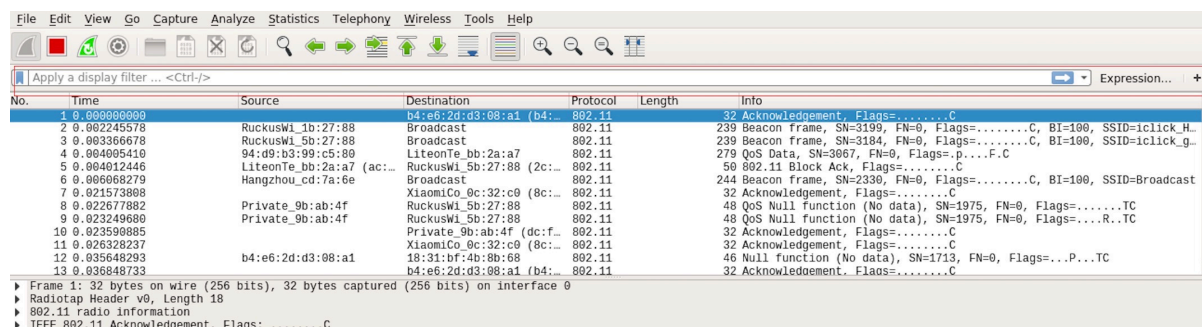


Fig. 64: Setting up Filters in Wireshark

Click *Filter*, the top left blue button in the picture below. The *display filter* dialogue box will appear.



Fig. 65: Display Filter Dialogue Box

Click the *Expression* button to bring up the *Filter Expression* dialogue box and set the filter according to your requirements.

The quickest way: enter the filters directly in the toolbar.

Click on this area to enter or modify the filters. If you enter a wrong or unfinished filter, the built-in syntax check turns the background red. As soon as the correct expression is entered, the background becomes green.

The previously entered filters are automatically saved. You can access them anytime by opening the drop down list.

For example, as shown in the picture below, enter two MAC addresses as the filters and click *Apply* (the blue arrow). In this case, only the packet data transmitted between these two MAC addresses will be captured.

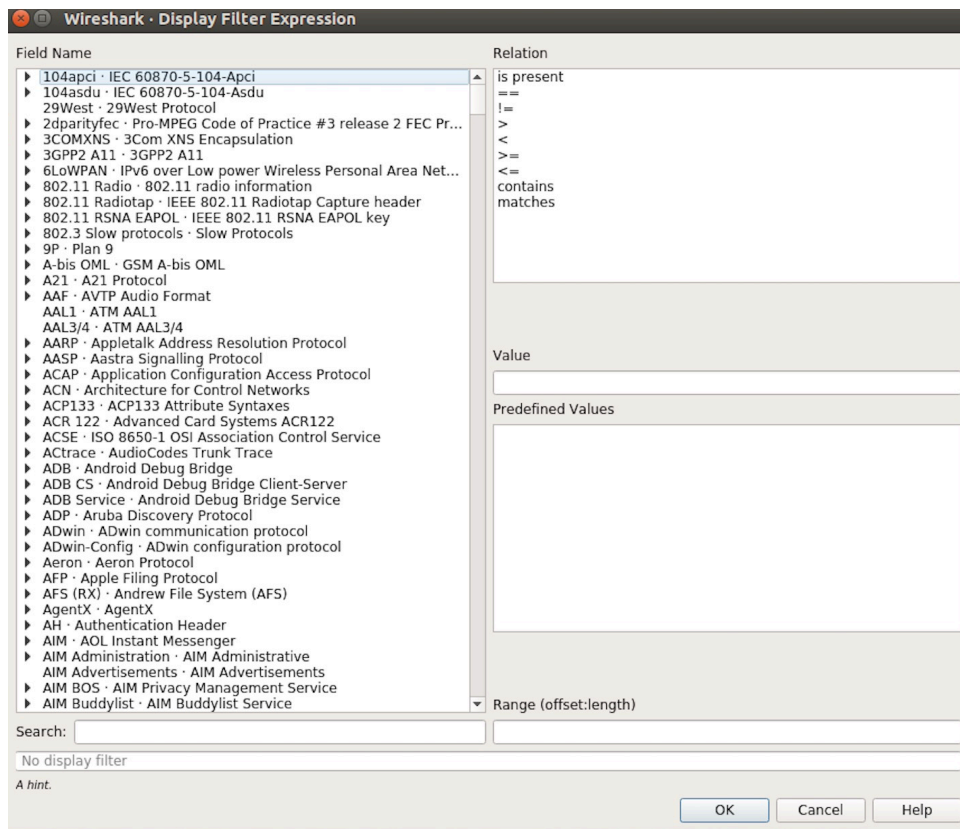


Fig. 66: Filter Expression Dialogue Box



Fig. 67: Filter Toolbar

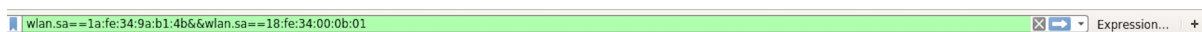


Fig. 68: Example of MAC Addresses applied in the Filter Toolbar

e) Packet List

You can click any packet in the packet list and check the detailed information about it in the box below the list. For example, if you click the first packet, its details will appear in that box.

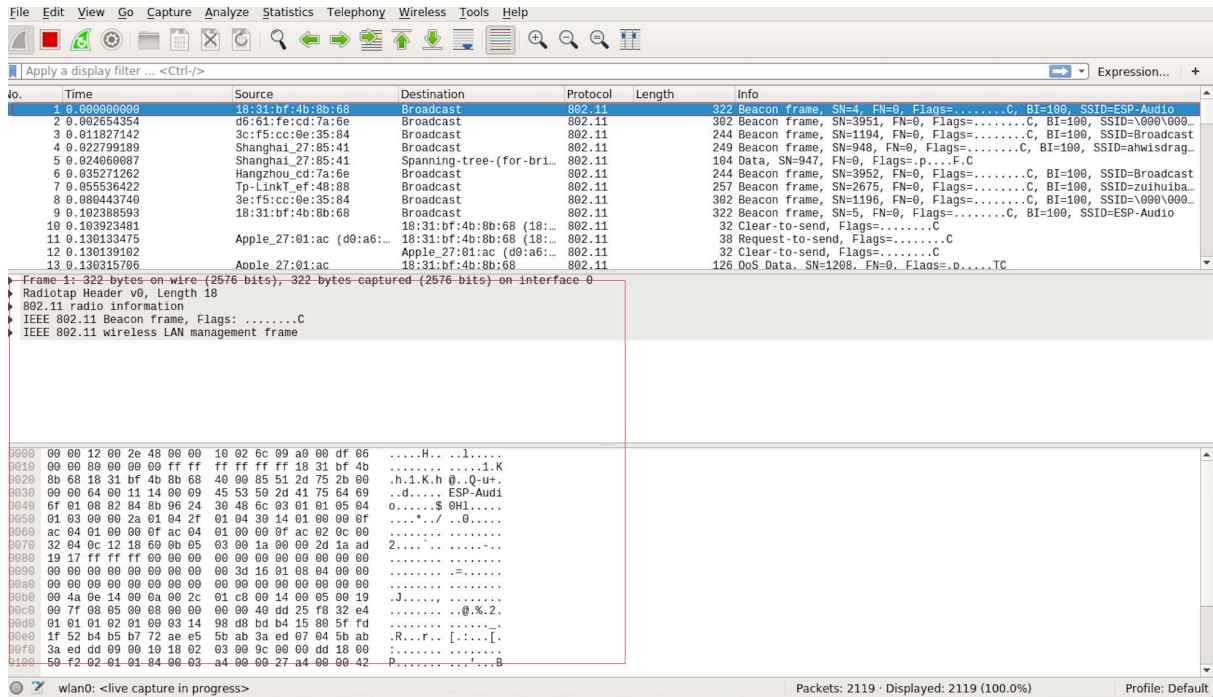


Fig. 69: Example of Packet List Details

f) Stop/Start Packet Capture

As shown in the picture below, click the red button to stop capturing the current packet.

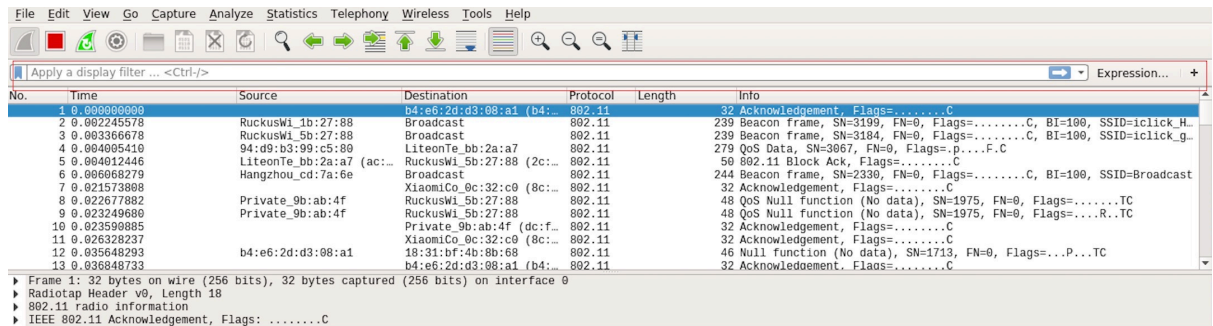


Fig. 70: Stopping Packet Capture

Click the top left blue button to start or resume packet capture.

g) Save the Current Packet

On Linux, go to *File -> Export Packet Dissections -> As Plain Text File* to save the packet.

Please note that *All packets, Displayed* and *All expanded* must be selected.

By default, Wireshark saves the captured packet in a libpcap file. You can also save the file in other formats, e.g. txt, to analyze it in other tools.

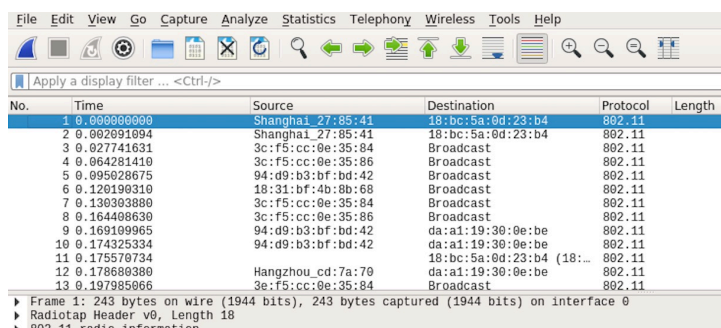


Fig. 71: Starting or Resuming the Packets Capture

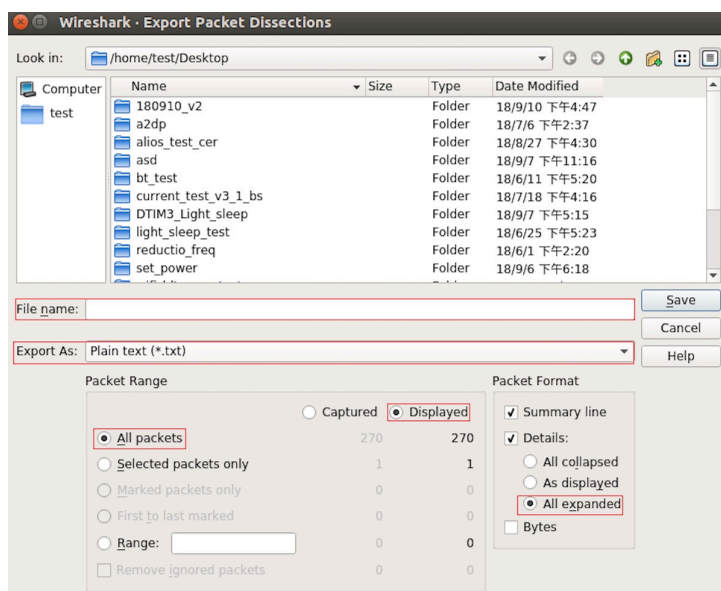


Fig. 72: Saving Captured Packets

Chapter 6

Libraries and Frameworks

6.1 Cloud Frameworks

ESP32 supports multiple cloud frameworks using agents built on top of ESP-IDF. Here are the pointers to various supported cloud frameworks' agents and examples:

6.1.1 AWS IoT

<https://github.com/espressif/esp-aws-iot> is an open source repository for ESP32 based on Amazon Web Services' `aws-iot-device-sdk-embedded-C`.

6.1.2 Azure IoT

<https://github.com/espressif/esp-azure> is an open source repository for ESP32 based on Microsoft Azure' s `azure-iot-sdk-c` SDK.

6.1.3 Google IoT Core

<https://github.com/espressif/esp-google-iot> is an open source repository for ESP32 based on Google' s `iot-device-sdk-embedded-c` SDK.

Chapter 7

Contributions Guide

We welcome contributions to the esp-idf project!

7.1 How to Contribute

Contributions to esp-idf - fixing bugs, adding features, adding documentation - are welcome. We accept contributions via [Github Pull Requests](#).

7.2 Before Contributing

Before sending us a Pull Request, please consider this list of points:

- Is the contribution entirely your own work, or already licensed under an Apache License 2.0 compatible Open Source License? If not then we unfortunately cannot accept it.
- Does any new code conform to the esp-idf [Style Guide](#)?
- Does the code documentation follow requirements in [Documenting Code](#)?
- Is the code adequately commented for people to understand how it is structured?
- Is there documentation or examples that go with code contributions? There are additional suggestions for writing good examples in [examples](#) readme.
- Are comments and documentation written in clear English, with no spelling or grammar errors?
- Example contributions are also welcome. Please check the [Creating Examples](#) guide for these.
- If the contribution contains multiple commits, are they grouped together into logical changes (one major change per pull request)? Are any commits with names like “fixed typo” [squashed into previous commits](#)?
- If you’re unsure about any of these points, please open the Pull Request anyhow and then ask us for feedback.

7.3 Pull Request Process

After you open the Pull Request, there will probably be some discussion in the comments field of the request itself.

Once the Pull Request is ready to merge, it will first be merged into our internal git system for in-house automated testing.

If this process passes, it will be merged onto the public github repository.

7.4 Legal Part

Before a contribution can be accepted, you will need to sign our [Contributor Agreement](#). You will be prompted for this automatically as part of the Pull Request process.

7.5 Related Documents

7.5.1 Espressif IoT Development Framework Style Guide

About This Guide

Purpose of this style guide is to encourage use of common coding practices within the ESP-IDF.

Style guide is a set of rules which are aimed to help create readable, maintainable, and robust code. By writing code which looks the same way across the code base we help others read and comprehend the code. By using same conventions for spaces and newlines we reduce chances that future changes will produce huge unreadable diffs. By following common patterns for module structure and by using language features consistently we help others understand code behavior.

We try to keep rules simple enough, which means that they can not cover all potential cases. In some cases one has to bend these simple rules to achieve readability, maintainability, or robustness.

When doing modifications to third-party code used in ESP-IDF, follow the way that particular project is written. That will help propose useful changes for merging into upstream project.

C Code Formatting

Indentation Use 4 spaces for each indentation level. Don't use tabs for indentation. Configure the editor to emit 4 spaces each time you press tab key.

Vertical Space Place one empty line between functions. Don't begin or end a function with an empty line.

```
void function1()
{
    do_one_thing();
    do_another_thing();
}
// INCORRECT, don't place empty line here
// place empty line here
void function2()
{
    // INCORRECT, don't use an empty line here
    int var = 0;
    while (var < SOME_CONSTANT) {
        do_stuff(&var);
    }
}
```

The maximum line length is 120 characters as long as it doesn't seriously affect the readability.

Horizontal Space Always add single space after conditional and loop keywords:

```
if (condition) { // correct
    // ...
}

switch (n) { // correct
    case 0:
        // ...
}

for(int i = 0; i < CONST; ++i) { // INCORRECT
    // ...
}
```

Add single space around binary operators. No space is necessary for unary operators. It is okay to drop space around multiply and divide operators:

```
const int y = y0 + (x - x0) * (y1 - y0) / (x1 - x0);    // correct
const int y = y0 + (x - x0)*(y1 - y0)/(x1 - x0);      // also okay
int y_cur = -y;                                       // correct
++y_cur;
const int y = y0+(x-x0)*(y1-y0)/(x1-x0);             // INCORRECT
```

No space is necessary around `.` and `->` operators.

Sometimes adding horizontal space within a line can help make code more readable. For example, you can add space to align function arguments:

```
gpio_matrix_in(PIN_CAM_D6,    I2S0I_DATA_IN14_IDX, false);
gpio_matrix_in(PIN_CAM_D7,    I2S0I_DATA_IN15_IDX, false);
gpio_matrix_in(PIN_CAM_HREF,  I2S0I_H_ENABLE_IDX,  false);
gpio_matrix_in(PIN_CAM_PCLK,  I2S0I_DATA_IN15_IDX, false);
```

Note however that if someone goes to add new line with a longer identifier as first argument (e.g. `PIN_CAM_VSYNC`), it will not fit. So other lines would have to be realigned, adding meaningless changes to the commit.

Therefore, use horizontal alignment sparingly, especially if you expect new lines to be added to the list later.

Never use TAB characters for horizontal alignment.

Never add trailing whitespace at the end of the line.

Braces

- Function definition should have a brace on a separate line:

```
// This is correct:
void function(int arg)
{
}

// NOT like this:
void function(int arg) {
}
```

- Within a function, place opening brace on the same line with conditional and loop statements:

```
if (condition) {
    do_one();
} else if (other_condition) {
    do_two();
}
```

Comments Use `//` for single line comments. For multi-line comments it is okay to use either `//` on each line or a `/* */` block.

Although not directly related to formatting, here are a few notes about using comments effectively.

- Don't use single comments to disable some functionality:

```
void init_something()
{
    setup_dma();
    // load_resources();           // WHY is this thing commented, asks_
    ↪the reader?
    start_timer();
}
```

- If some code is no longer required, remove it completely. If you need it you can always look it up in git history of this file. If you disable some call because of temporary reasons, with an intention to restore it in the future, add explanation on the adjacent line:

```
void init_something()
{
    setup_dma();
    // TODO: we should load resources here, but loader is not fully integrated_
    ↪yet.
    // load_resources();
    start_timer();
}
```

- Same goes for #if 0 ... #endif blocks. Remove code block completely if it is not used. Otherwise, add comment explaining why the block is disabled. Don't use #if 0 ... #endif or comments to store code snippets which you may need in the future.
- Don't add trivial comments about authorship and change date. You can always look up who modified any given line using git. E.g. this comment adds clutter to the code without adding any useful information:

```
void init_something()
{
    setup_dma();
    // XXX add 2016-09-01
    init_dma_list();
    fill_dma_item(0);
    // end XXX add
    start_timer();
}
```

Line Endings Commits should only contain files with LF (Unix style) endings.

Windows users can configure git to check out CRLF (Windows style) endings locally and commit LF endings by setting the `core.autocrlf` setting. *Github has a document about setting this option* <[github-line-endings](#)>. However because MSYS2 uses Unix-style line endings, it is often easier to configure your text editor to use LF (Unix style) endings when editing ESP-IDF source files.

If you accidentally have some commits in your branch that add LF endings, you can convert them to Unix by running this command in an MSYS2 or Unix terminal (change directory to the IDF working directory and check the correct branch is currently checked out, beforehand):

```
git rebase --exec 'git diff-tree --no-commit-id --name-only -r HEAD | xargs_
    ↪dos2unix && git commit -a --amend --no-edit --allow-empty' master
```

(Note that this line rebases on master, change the branch name at the end to rebase on another branch.)

For updating a single commit, it's possible to run `dos2unix FILENAME` and then run `git commit --amend`

Formatting Your Code You can use `astyle` program to format your code according to the above recommendations.

If you are writing a file from scratch, or doing a complete rewrite, feel free to re-format the entire file. If you are changing a small portion of file, don't re-format the code you didn't change. This will help others when they review your changes.

To re-format a file, run:

```
tools/format.sh components/my_component/file.c
```

C++ Code Formatting

The same rules as for C apply. Where they are not enough, apply the following rules.

File Naming C++ Header files have the extension `.h`. C++ source files have the extension `.cpp`, which is important for the compiler to distinguish them from normal C source files.

Naming

- **Class and struct** names shall be written in `CamelCase` with a capital letter as beginning. Member variables and methods shall be in `snake_case`.
- **Namespaces** shall be in lower `snake_case`.
- **Templates** are specified in the line above the function declaration.

Member Order in Classes First put the public members, then the protected, then private ones. Omit public, protected or private sections without any members.

Spacing

- Don't indent inside namespaces.
- Put public, protected and private labels at the same indentation level as the corresponding class label.

Simple Example

```
// file spaceship.h
#ifndef SPACESHIP_H_
#define SPACESHIP_H_
#include <cstdlib>

namespace spaceships {

class SpaceShip {
public:
    SpaceShip(size_t crew);
    size_t get_crew_size() const;

private:
    const size_t crew;
};

class SpaceShuttle : public SpaceShip {
public:
    SpaceShuttle();
};

class Sojuz : public SpaceShip {
public:
    Sojuz();
};

template <typename T>
class CargoShip {
```

(continues on next page)


```
public:
    CargoShip(const T &cargo);

private:
    T cargo;
};

} // namespace spaceships

#endif // SPACESHIP_H_

// file spaceship.cpp
#include "spaceship.h"

namespace spaceships {

// Putting the curly braces in the same line for constructors is OK if it only
// initializes
// values in the initializer list
SpaceShip::SpaceShip(size_t crew) : crew(crew) { }

size_t SpaceShip::get_crew_size() const
{
    return crew;
}

SpaceShuttle::SpaceShuttle() : SpaceShip(7)
{
    // doing further initialization
}

Sojuz::Sojuz() : SpaceShip(3)
{
    // doing further initialization
}

template <typename T>
CargoShip<T>::CargoShip(const T &cargo) : cargo(cargo) { }

} // namespace spaceships
```

CMake Code Style

- Indent with four spaces.
- Maximum line length 120 characters. When splitting lines, try to focus on readability where possible (for example, by pairing up keyword/argument pairs on individual lines).
- Don't put anything in the optional parentheses after `endforeach()`, `endif()`, etc.
- Use lowercase (`with_underscores`) for command, function, and macro names.
- For locally scoped variables, use lowercase (`with_underscores`).
- For globally scoped variables, use uppercase (`WITH_UNDERSCORES`).
- Otherwise follow the defaults of the [cmake-lint](#) project.

Configuring the Code Style for a Project Using EditorConfig

EditorConfig helps developers define and maintain consistent coding styles between different editors and IDEs. The EditorConfig project consists of a file format for defining coding styles and a collection of text editor plugins that enable editors to read the file format and adhere to defined styles. EditorConfig files are easily readable and they work nicely with version control systems.

For more information, see [EditorConfig Website](#).

Documenting Code

Please see the guide here: [Documenting Code](#).

Naming

- Any variable or function which is only used in a single source file should be declared `static`.
- Public names (non-static variables and functions) should be namespaced with a per-component or per-unit prefix, to avoid naming collisions. ie `esp_vfs_register()` or `esp_console_run()`. Starting the prefix with `esp_` for Espressif-specific names is optional, but should be consistent with any other names in the same component.
- Static variables should be prefixed with `s_` for easy identification. For example, `static bool s_invert`.
- Avoid unnecessary abbreviations (ie shortening `data` to `dat`), unless the resulting name would otherwise be very long.

Structure

To be written.

Language Features

To be written.

7.5.2 Documenting Code

The purpose of this description is to provide quick summary on documentation style used in [espressif/esp-idf](#) repository and how to add new documentation.

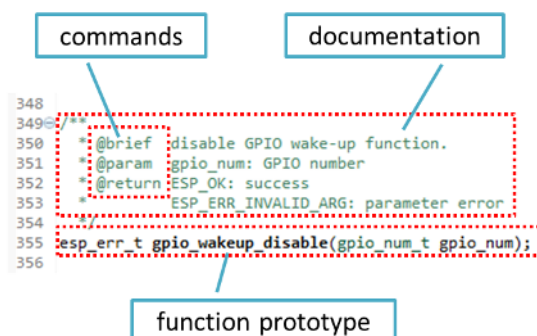
Introduction

When documenting code for this repository, please follow [Doxygen style](#). You are doing it by inserting special commands, for instance `@param`, into standard comments blocks, for example:

```
/**
 * @param ratio this is oxygen to air ratio
 */
```

Doxygen is phrasing the code, extracting the commands together with subsequent text, and building documentation out of it.

Typical comment block, that contains documentation of a function, looks like below.

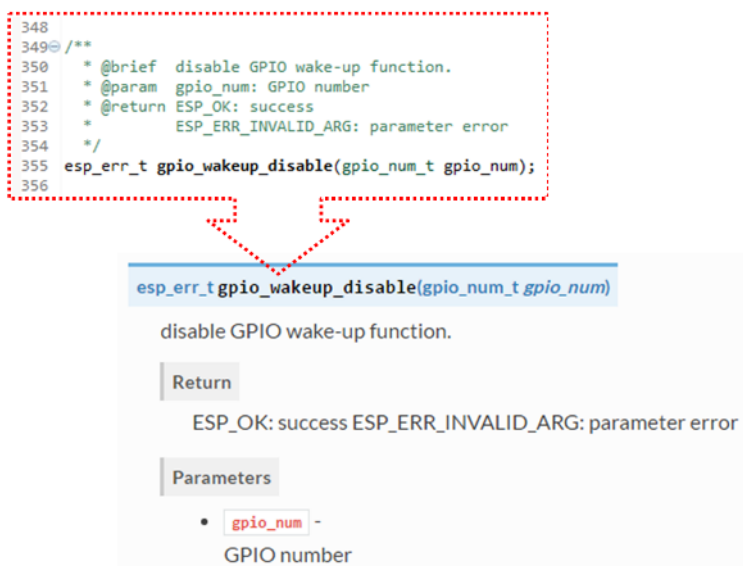


Doxygen supports couple of formatting styles. It also gives you great flexibility on level of details to include in documentation. To get familiar with available features, please check data rich and very well organized [Doxygen Manual](#).

Why we need it?

The ultimate goal is to ensure that all the code is consistently documented, so we can use tools like [Sphinx](#) and [Breathe](#) to aid preparation and automatic updates of API documentation when the code changes.

With these tools the above piece of code renders like below:



Go for it!

When writing code for this repository, please follow guidelines below.

1. Document all building blocks of code: functions, structs, typedefs, enums, macros, etc. Provide enough information on purpose, functionality and limitations of documented items, as you would like to see them documented when reading the code by others.
2. Documentation of function should describe what this function does. If it accepts input parameters and returns some value, all of them should be explained.
3. Do not add a data type before parameter or any other characters besides spaces. All spaces and line breaks are compressed into a single space. If you like to break a line, then break it twice.

```

41 @/**
42  * @brief Set log level for given tag
43  *
44  * If logging for given component has already been enabled, changes previous setting.
45  *
46  * @param tag Tag of the log entries to enable. Must be a non-NULL zero terminated string.
47  *           Value "" resets log level for all tags to the given value.
48  *
49  * @param level Selects log level to enable.
50  *             Only logs at this and lower levels will be shown.
51  */
52 void esp_log_level_set(const char* tag, esp_log_level_t level);
    
```

do not add data type

white spaces are compressed

a line break that will render

this line break will not render

```
void esp_log_level_set(const char* tag, esp_log_level_t level)
```

Set log level for given tag.

If logging for given component has already been enabled, changes previous setting.

Parameters

- tag** - Tag of the log entries to enable. Must be a non-NULL zero terminated string. Value "" resets log level for all tags to the given value.
- level** - Selects log level to enable. Only logs at this and lower levels will be shown.

4. If function has void input or does not return any value, then skip @param or @return

```

26 @/**
27  * @brief Initialize BT controller
28  *
29  * This function should be called only once,
30  * before any other BT functions are called.
31  */
32 void bt_controller_init(void);
    
```

```
void bt_controller_init(void)
```

Initialize BT controller.

This function should be called only once, before any other BT functions are called.

5. When documenting a define as well as members of a struct or enum, place specific comment like below after each member.

```

45 @/**
46  * Mode of opening the non-volatile storage
47  *
48  */
49 @typedef enum {
50     NVS_READONLY, /*!< Read only */
51     NVS_READWRITE /*!< Read and write */
52 } nvs_open_mode;
    
```

enum nvs_open_mode

Mode of opening the non-volatile storage.

Values:

- NVS_READONLY**
Read only
- NVS_READWRITE**
Read and write

/*!< how to documented members */

6. To provide well formatted lists, break the line after command (like @return in example below).

```

*
* @return
*   - ESP_OK if erase operation was successful
    
```

(continues on next page)

(continued from previous page)

```
*
*   - ESP_ERR_NVS_INVALID_HANDLE if handle has been closed or is NULL
*   - ESP_ERR_NVS_READ_ONLY if handle was opened as read only
*   - ESP_ERR_NVS_NOT_FOUND if the requested key doesn't exist
*   - other error codes from the underlying storage driver
*
```

7. Overview of functionality of documented header file, or group of files that make a library, should be placed in the same directory in a separate README.rst file. If directory contains header files for different APIs, then the file name should be apiname-readme.rst.

Go one extra mile

There is couple of tips, how you can make your documentation even better and more useful to the reader.

1. Add code snippets to illustrate implementation. To do so, enclose snippet using @code{c} and @endcode commands.

```
*
* @code{c}
* // Example of using nvs_get_i32:
* int32_t max_buffer_size = 4096; // default value
* esp_err_t err = nvs_get_i32(my_handle, "max_buffer_size", &max_buffer_size);
* assert(err == ESP_OK || err == ESP_ERR_NVS_NOT_FOUND);
* // if ESP_ERR_NVS_NOT_FOUND was returned, max_buffer_size will still
* // have its default value.
* @endcode
*
```

The code snippet should be enclosed in a comment block of the function that it illustrates.

2. To highlight some important information use command @attention or @note.

```
*
* @attention
*   1. This API only impact WIFI_MODE_STA or WIFI_MODE_APSTA mode
*   2. If the ESP32 is connected to an AP, call esp_wifi_disconnect to
*   ↪disconnect.
*
```

Above example also shows how to use a numbered list.

3. To provide common description to a group of similar functions, enclose them using /**@{ */ and /**@} */ markup commands:

```
/**@{ */
/**
* @brief common description of similar functions
*
*/
void first_similar_function (void);
void second_similar_function (void);
/**@} */
```

For practical example see [nvs_flash/include/nvs.h](#).

4. You may want to go even further and skip some code like e.g. repetitive defines or enumerations. In such case enclose the code within /** @cond */ and /** @endcond */ commands. Example of such implementation is provided in [driver/include/driver/gpio.h](#).
5. Use markdown to make your documentation even more readable. You will add headers, links, tables and more.

```
*
* [ESP32 Technical Reference Manual](https://espressif.com/sites/default/files/
* ↪documentation/esp32_technical_reference_manual_en.pdf)
*
```

Note: Code snippets, notes, links, etc. will not make it to the documentation, if not enclosed in a comment block associated with one of documented objects.

6. Prepare one or more complete code examples together with description. Place description in a separate file `README.md` in specific folder of `examples` directory.

Linking Examples

When linking to examples on GitHub do not use absolute / hardcoded URLs. Instead, use docutils custom roles that will generate links for you. These auto-generated links point to the tree or blob for the git commit ID (or tag) of the repository. This is needed to ensure that links do not get broken when files in master branch are moved around or deleted.

The following roles are provided:

- `:idf:`path`` - points to directory inside ESP-IDF
- `:idf_file:`path`` - points to file inside ESP-IDF
- `:idf_raw:`path`` - points to raw view of the file inside ESP-IDF
- `:component:`path`` - points to directory inside ESP-IDF components dir
- `:component_file:`path`` - points to file inside ESP-IDF components dir
- `:component_raw:`path`` - points to raw view of the file inside ESP-IDF components dir
- `:example:`path`` - points to directory inside ESP-IDF examples dir
- `:example_file:`path`` - points to file inside ESP-IDF examples dir
- `:example_raw:`path`` - points to raw view of the file inside ESP-IDF examples dir

Example implementation:

```
* :example:`get-started/hello_world`
* :example:`Hello World! <get-started/hello_world>`
```

How it renders:

- [get-started/hello_world](#)
- [Hello World!](#)

A check is added to the CI build script, which searches RST files for presence of hard-coded links (identified by `tree/master`, `blob/master`, or `raw/master` part of the URL). This check can be run manually: `cd docs` and then `make gh-linkcheck`.

Linking Language Versions

Switching between documentation in different languages may be done using `:link_to_translation:` custom role. The role placed on a page of documentation provides a link to the same page in a language specified as a parameter. Examples below show how to enter links to Chinese and English versions of documentation:

```
:link_to_translation:`zh_CN: 中文版`
:link_to_translation:`en:English`
```

The language is specified using standard abbreviations like `en` or `zh_CN`. The text after last semicolon is not standardized and may be entered depending on the context where the link is placed, e.g.:

```
:link_to_translation:`en:see description in English`
```

Add Illustrations

Consider adding diagrams and pictures to illustrate described concepts.

Sometimes it is better to add an illustration than writing a lengthy paragraph to describe a complex idea, a data structure or an algorithm. This repository is using [blockdiag](#) suite of tools to generate diagram images from simple text files.

The following types of diagrams are supported:

- [Block diagram](#)
- [Sequence diagram](#)
- [Activity diagram](#)
- [Logical network diagram](#)

With this suite of tools it is possible to generate beautiful diagram images from simple text format (similar to `graphviz`'s DOT format). The diagram elements are laid out automatically. The diagram code is then converted into “.png” graphics and integrated “behind the scenes” into **Sphinx** documents.

For the diagram preparation you can use an on-line [interactive shell](#) that instantly shows the rendered image.

Below are couple of diagram examples:

- Simple **block diagram** / `blockdiag` - [Wi-Fi Buffer Configuration](#)
- Slightly more complicated **block diagram** - [Wi-Fi programming model](#)
- **Sequence diagram** / `seqdiag` - [Scan for a Specific AP in All Channels](#)
- **Packet diagram** / `packetdiag` - [NVS Page Structure](#)

Try them out by modifying the source code and see the diagram instantly rendering below.

Note: There may be slight differences in rendering of font used by the [interactive shell](#) compared to the font used in the esp-idf documentation.

Add Notes

Working on a document, you might need to:

- Place some suggestions on what should be added or modified in future.
- Leave a reminder for yourself or somebody else to follow up.

In this case, add a todo note to your reST file using the directive `.. todo::`. For example:

```
.. todo::  
  
    Add a package diagram.
```

If you add `.. todoclist::` to a reST file, the directive will be replaced by a list of all todo notes from the whole documentation.

By default, the directives `.. todo::` and `.. todoclist::` are ignored by documentation builders. If you want the notes and the list of notes to be visible in your locally built documentation, do the following:

1. Open your local `conf_common.py` file.
2. Find the parameter `todo_include_todos`.
3. Change its value from `False` to `True`.

Before pushing your changes to origin, please set the value of `todo_include_todos` back to `False`.

For more details about the extension, see [sphinx.ext.todo](#) documentation.

Put it all together

Once documentation is ready, follow instruction in [API Documentation Template](#) and create a single file, that will merge all individual pieces of prepared documentation. Finally add a link to this file to respective `.. toctree::` in `index.rst` file located in `/docs` folder or subfolders.

OK, but I am new to Sphinx!

1. No worries. All the software you need is well documented. It is also open source and free. Start by checking [Sphinx](#) documentation. If you are not clear how to write using rst markup language, see [reStructuredText Primer](#). You can also use markdown (.md) files, and find out more about the specific markdown syntax that we use on [Recommonmark parser](#)'s documentation page <<https://recommonmark.readthedocs.io/en/latest/>>`_`.
2. Check the source files of this documentation to understand what is behind of what you see now on the screen. Sources are maintained on GitHub in [espressif/esp-idf](#) repository in [docs](#) folder. You can go directly to the source file of this page by scrolling up and clicking the link in the top right corner. When on GitHub, see what's really inside, open source files by clicking [Raw](#) button.
3. You will likely want to see how documentation builds and looks like before posting it on the GitHub. There are two options to do so:
 - Install [Sphinx](#), [Breathe](#), [Blockdiag](#) and [Doxygen](#) to build it locally, see chapter below.
 - Set up an account on [Read the Docs](#) and build documentation in the cloud. Read the Docs provides document building and hosting for free and their service works really quick and great.
4. To preview documentation before building, use [Sublime Text](#) editor together with [OmniMarkupPreviewer](#) plugin.

Setup for building documentation locally

You can setup environment to build documentation locally on your PC by installing:

1. Doxygen - <https://www.stack.nl/~dimitri/doxygen/>
2. Sphinx - <https://github.com/sphinx-doc/sphinx/#readme-for-sphinx>
3. Breathe - <https://github.com/michaeljones/breathe#breathe>
4. Document theme "sphinx_rtd_theme" - https://github.com/rtfd/sphinx_rtd_theme
5. Custom 404 page "sphinx-notfound-page" - <https://github.com/rtfd/sphinx-notfound-page>
6. Blockdiag - <http://blockdiag.com/en/index.html>
7. Recommonmark - <https://github.com/rtfd/recommonmark>

The package "sphinx_rtd_theme" is added to have the same "look and feel" of [ESP32 Programming Guide](#) documentation like on the "Read the Docs" hosting site.

Do not worry about being confronted with several packages to install. Besides Doxygen, all remaining packages are written in Python. Therefore installation of all of them is combined into one simple step.

Installation of Doxygen is OS dependent:

Linux

```
sudo apt-get install doxygen
```

Windows - install in MSYS2 console

```
pacman -S doxygen
```

MacOS

```
brew install doxygen
```

Note: If you are installing on Windows system (Linux and MacOS users should skip this note), **before** going further, execute two extra steps below. These steps are required to install dependencies of "blockdiag" discussed under [Add Illustrations](#).

1. Update all the system packages:

```
$ pacman -Syu
```

This process will likely require restarting of the MSYS2 MINGW32 console and repeating above commands, until update is complete.

2. Install *pillow*, that is one of dependences of the *blockdiag*:


```
$ pacman -S mingw32/mingw-w64-i686-python2-pillow
```

Check the log on the screen that `mingw-w64-i686-python2-pillow-4.3.0-1` is installed. Previous versions of *pillow* will not work.

A downside of Windows installation is that fonts of the *blockdiag* pictures *<add-illustrations>* do not render correctly, you will see some random characters instead. Until this issue is fixed, you can use the [interactive shell](#) to see how the complete picture looks like.

All remaining applications are [Python](#) packages and you can install them in one step as follows:

```
cd ~/esp/esp-idf/docs
pip install --user -r requirements.txt
```

Note: Installation steps assume that ESP-IDF is placed in `~/esp/esp-idf` directory, that is default location of ESP-IDF used in documentation.

Change to directory with files for specific language:

```
cd en
```

Now you should be ready to build documentation by invoking:

```
make html
```

This may take couple of minutes. After completion, documentation will be placed in `~/esp/esp-idf/docs/en/_build/html` folder. To see it, open `index.html` in a web browser.

Wrap up

We love good code that is doing cool things. We love it even better, if it is well documented, so we can quickly make it run and also do the cool things.

Go ahead, contribute your code and documentation!

Related Documents

- [API Documentation Template](#)
- [Documentation Add-ons and Extensions Reference](#)

7.5.3 Documentation Add-ons and Extensions Reference

This documentation is created using [Sphinx](#) application that renders text source files in [reStructuredText](#) (`.rst`) format located in `docs` directory. For some more details on that process, please refer to section [Documenting Code](#).

Besides Sphinx there are several other applications that help to provide nicely formatted and easy to navigate documentation. These applications are listed in section [Setup for building documentation locally](#) with the installed version numbers provided in file `docs/requirements.txt`.

On top of that we have created a couple of custom add-ons and extensions to help integrate documentation with underlying [ESP-IDF](#) repository and further improve navigation as well as maintenance of documentation.

The purpose of this section is to provide a quick reference to the add-ons and the extensions.

Documentation Folder Structure

- The ESP-IDF repository contains a dedicated documentation folder `docs` in the root.
- The `docs` folder contains localized documentation in `docs/en` (English) and `docs/zh_CN` (simplified Chinese) subfolders.
- Graphics files and fonts common to localized documentation are contained in `docs/_static` subfolder
- Remaining files in the root of `docs` as well as `docs/en` and `docs/zh_CN` provide configuration and scripts used to automate documentation processing including the add-ons and extensions.
- Several folders and files are generated dynamically during documentations build and placed primarily in `docs/[lang]/_build` folders. These folders are temporary and not visible in ESP-IDF repository,

Add-ons and Extensions Reference

docs/conf_common.py This file contains configuration common to each localized documentation (e.g. English, Chinese). The contents of this file is imported to standard Sphinx configuration file `conf.py` located in respective language folders (e.g. `docs/en`, `docs/zh_CN`) during build for each language.

docs/check_doc_warnings.sh If there are any warnings reported during documentation build, then the build is failed. The warnings should be resolved before merging any documentation updates. This script is doing check for warnings in respective log file to fail the build. See also description of `sphinx-known-warnings.txt` below.

docs/check_lang_folder_sync.sh To reduce potential discrepancies when maintaining concurrent language version, the structure and filenames of language folders `docs/en` and `docs/zh_CN` folders should be kept identical. The script `check_lang_folder_sync.sh` is run on each documentation build to verify if this condition is met.

Note: If a new content is provided in e.g. English, and there is no any translation yet, then the corresponding file in `zh_CN` folder should contain an `.. include::` directive pointing to the source file in English. This will automatically include the English version visible to Chinese readers. For example if a file `docs/zh_CN/contribute/documenting-code.rst` does not have a Chinese translation, then it should contain `.. include:: ../../en/contribute/documenting-code.rst` instead.

docs/docs_common.mk It contains the common code which is included into the language-specific `Makefiles`. Note that this file contains couple of customizations comparing to what is provided within standard Sphinx installation, e.g. `gh-linkcheck` command has been added.

docs/gen-dxd.py A Python script that generates API reference files based on Doxygen `xml` output. The files have an `inc` extension and are located in `docs/[lang]/_build/inc` directory created dynamically when documentation is build. Please refer to *Documenting Code* and *API Documentation Template*, section **API Reference** for additional details on this process.

docs/gen-toolchain-links.py There couple of places in documentation that provide links to download the toolchain. To provide one source of this information and reduce effort to manually update several files, this script generates toolchain download links and toolchain unpacking code snippets based on information found in `tools/toolchain_versions.mk`.

docs/gen-version-specific-includes.py Another Python script to automatically generate reStructuredText `Text .inc` snippets with version-based content for this ESP-IDF version.

docs/html_redirects.py During documentation lifetime some source files are moved between folders or renamed. This Python script is adding a mechanism to redirect documentation pages that have changed URL by generating in the Sphinx output static HTML redirect pages. The script is used together with a redirection list `html_redirect_pages` defined in file `docs/conf_common.py`.

docs/link-roles.py This is an implementation of a custom [Sphinx Roles](#) to help linking from documentation to specific files and folders in ESP-IDF. For description of implemented roles please see *Linking Examples* and *Linking Language Versions*.

docs/local_util.py A collection of utility functions useful primarily when building documentation locally (see *Setup for building documentation locally*) to reduce the time to generate documentation on a second and subsequent builds. The utility functions check what Doxygen `xml` input files have been changed and copy these files to destination folders, so only the changed files are used during build process.

docs/sphinx-known-warnings.txt There are couple of spurious Sphinx warnings that cannot be resolved without doing update to the Sphinx source code itself. For such specific cases respective warnings are documented in

`sphinx-known-warnings.txt` file, that is checked during documentation build, to ignore the spurious warnings.

tools/gen_esp_err_to_name.py This script is traversing the ESP-IDF directory structure looking for error codes and messages in source code header files to generate an `.inc` file to include in documentation under *Error Codes Reference*.

tools/kconfig_new/confgen.py Options to configure ESP-IDF's components are contained in `Kconfig` files located inside directories of individual components, e.g. `components/bt/Kconfig`. This script is traversing the `component` directories to collect configuration options and generate an `.inc` file to include in documentation under *Configuration Options Reference*.

Related Documents

- [Documenting Code](#)

7.5.4 Creating Examples

Each ESP-IDF example is a complete project that someone else can copy and adapt the code to solve their own problem. Examples should demonstrate ESP-IDF functionality, while keeping this purpose in mind.

Structure

- The `main` directory should contain a source file named `(something)_example_main.c` with the main functionality.
- If the example has additional functionality, split it logically into separate C or C++ source files under `main` and place a corresponding header file in the same directory.
- If the example has a lot of additional functionality, consider adding a `components` directory to the example project and make some example-specific components with library functionality. Only do this if the components are specific to the example, if they're generic or common functionality then they should be added to ESP-IDF itself.
- The example should have a `README.md` file. Use the [template example README](#) and adapt it for your particular example.
- Examples should have an `example_test.py` file for running an automated example test. If submitting a GitHub Pull Request which includes an example, it's OK not to include this file initially. The details can be discussed as part of the [Pull Request](#).

General Guidelines

Example code should follow the [Espressif IoT Development Framework Style Guide](#).

Checklist

Checklist before submitting a new example:

- Example project name (in `Makefile` and `README.md`) uses the word “example”. Use “example” instead of “demo”, “test” or similar words.
- Example does one distinct thing. If the example does more than one thing at a time, split it into two or more examples.
- Example has a `README.md` file which is similar to the [template example README](#).
- Functions and variables in the example are named according to [naming section of the style guide](#). (For non-static names which are only specific to the example's source files, you can use `example` or something similar as a prefix.)
- All code in the example is well structured and commented.
- Any unnecessary code (old debugging logs, commented-out code, etc.) is removed from the example.
- Options in the example (like network names, addresses, etc) are not hard-coded. Use configuration items if possible, or otherwise declare macros or constants)

- Configuration items are provided in a `KConfig.projbuild` file with a menu named “Example Configuration”. See existing example projects to see how this is done.
- All original example code has a license header saying it is “in the public domain / CC0”, and a warranty disclaimer clause. Alternatively, the example is licensed under Apache License 2.0. See existing examples for headers to adapt from.
- Any adapted or third party example code has the original license header on it. This code must be licensed compatible with Apache License 2.0.

7.5.5 API Documentation Template

Note: INSTRUCTIONS

1. Use this file ([docs/api-reference/template.rst](#)) as a template to document API.
 2. Change the file name to the name of the header file that represents documented API.
 3. Include respective files with descriptions from the API folder using `..include::`
 - `README.rst`
 - `example.rst`
 - ...
 4. Optionally provide description right in this file.
 5. Once done, remove all instructions like this one and any superfluous headers.
-

Overview

Note: INSTRUCTIONS

1. Provide overview where and how this API may be used.
 2. Where applicable include code snippets to illustrate functionality of particular functions.
 3. To distinguish between sections, use the following [heading levels](#):
 - `#` with overline, for parts
 - `*` with overline, for chapters
 - `=`, for sections
 - `-`, for subsections
 - `^`, for subsubsections
 - `"`, for paragraphs
-

Application Example

Note: INSTRUCTIONS

1. Prepare one or more practical examples to demonstrate functionality of this API.
 2. Each example should follow pattern of projects located in `esp-idf/examples/` folder.
 3. Place example in this folder complete with `README.md` file.
 4. Provide overview of demonstrated functionality in `README.md`.
 5. With good overview reader should be able to understand what example does without opening the source code.
 6. Depending on complexity of example, break down description of code into parts and provide overview of functionality of each part.
 7. Include flow diagram and screenshots of application output if applicable.
 8. Finally add in this section synopsis of each example together with link to respective folder in `esp-idf/examples/`.
-

API Reference

Note: INSTRUCTIONS

1. This repository provides for automatic update of API reference documentation using *code markup retrieved by Doxygen from header files*.
2. Update is done on each documentation build by invoking script `docs/gen-dxd.py` for all header files listed in the INPUT statement of `docs/Doxyfile`.
3. Each line of the INPUT statement (other than a comment that begins with `##`) contains a path to header file `*.h` that will be used to generate corresponding `*.inc` files:

```
##
## Wi-Fi - API Reference
##
../components/esp32/include/esp_wifi.h \
../components/esp32/include/esp_smartconfig.h \
```

4. The `*.inc` files contain formatted reference of API members generated automatically on each documentation build. All `*.inc` files are placed in Sphinx `_build` directory. To see directives generated for e.g. `esp_wifi.h`, run `python gen-dxd.py esp32/include/esp_wifi.h`.
5. To show contents of `*.inc` file in documentation, include it as follows:

```
.. include:: ../_build/inc/esp_wifi.inc
```

For example see `docs/en/api-reference/wifi/esp_wifi.rst`

6. Optionally, rather than using `*.inc` files, you may want to describe API in your own way. See `docs/en/api-guides/ulp.rst` for example.

Below is the list of common `.. doxygen...::` directives:

- Functions - `.. doxygenfunction:: name_of_function`
- Unions - `.. doxygenunion:: name_of_union`
- Structures - `.. doxygenstruct:: name_of_structure` together with `:members:`
- Macros - `.. doxygendefine:: name_of_define`
- Type Definitions - `.. doxygentypedef:: name_of_type`
- Enumerations - `.. doxygenenum:: name_of_enumeration`

See [Breathe documentation](#) for additional information.

To provide a link to header file, use the *link custom role* as follows:

```
* :component_file:`path_to/header_file.h`
```

7. In any case, to generate API reference, the file `docs/Doxyfile` should be updated with paths to `*.h` headers that are being documented.
8. When changes are committed and documentation is built, check how this section has been rendered. *Correct annotations* in respective header files, if required.

7.5.6 Contributor Agreement

Individual Contributor Non-Exclusive License Agreement

including the Traditional Patent License OPTION

Thank you for your interest in contributing to Espressif IoT Development Framework (esp-idf) (“We” or “Us”). The purpose of this contributor agreement (“Agreement”) is to clarify and document the rights granted by contributors to Us. To make this document effective, please follow the instructions at [CONTRIBUTING.rst](#)

1. DEFINITIONS “You” means the Individual Copyright owner who submits a Contribution to Us. If You are an employee and submit the Contribution as part of your employment, You have had Your employer approve this Agreement or sign the Entity version of this document.

“**Contribution**” means any original work of authorship (software and/or documentation) including any modifications or additions to an existing work, Submitted by You to Us, in which You own the Copyright. If You do not own the Copyright in the entire work of authorship, please contact Us at angus@espressif.com.

“**Copyright**” means all rights protecting works of authorship owned or controlled by You, including copyright, moral and neighboring rights, as appropriate, for the full term of their existence including any extensions by You.

“**Material**” means the software or documentation made available by Us to third parties. When this Agreement covers more than one software project, the Material means the software or documentation to which the Contribution was Submitted. After You Submit the Contribution, it may be included in the Material.

“**Submit**” means any form of physical, electronic, or written communication sent to Us, including but not limited to electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, Us, but excluding communication that is conspicuously marked or otherwise designated in writing by You as “Not a Contribution.”

“**Submission Date**” means the date You Submit a Contribution to Us.

“**Documentation**” means any non-software portion of a Contribution.

2. LICENSE GRANT 2.1 Copyright License to Us

Subject to the terms and conditions of this Agreement, You hereby grant to Us a worldwide, royalty-free, NON-exclusive, perpetual and irrevocable license, with the right to transfer an unlimited number of non-exclusive licenses or to grant sublicenses to third parties, under the Copyright covering the Contribution to use the Contribution by all means, including, but not limited to:

- to publish the Contribution,
- to modify the Contribution, to prepare derivative works based upon or containing the Contribution and to combine the Contribution with other software code,
- to reproduce the Contribution in original or modified form,
- to distribute, to make the Contribution available to the public, display and publicly perform the Contribution in original or modified form.

2.2 Moral Rights remain unaffected to the extent they are recognized and not waivable by applicable law. Notwithstanding, You may add your name in the header of the source code files of Your Contribution and We will respect this attribution when using Your Contribution.

3. PATENTS 3.1 Patent License

Subject to the terms and conditions of this Agreement You hereby grant to us a worldwide, royalty-free, non-exclusive, perpetual and irrevocable (except as stated in Section 3.2) patent license, with the right to transfer an unlimited number of non-exclusive licenses or to grant sublicenses to third parties, to make, have made, use, sell, offer for sale, import and otherwise transfer the Contribution and the Contribution in combination with the Material (and portions of such combination). This license applies to all patents owned or controlled by You, whether already acquired or hereafter acquired, that would be infringed by making, having made, using, selling, offering for sale, importing or otherwise transferring of Your Contribution(s) alone or by combination of Your Contribution(s) with the Material.

3.2 Revocation of Patent License

You reserve the right to revoke the patent license stated in section 3.1 if we make any infringement claim that is targeted at your Contribution and not asserted for a Defensive Purpose. An assertion of claims of the Patents shall be considered for a “Defensive Purpose” if the claims are asserted against an entity that has filed, maintained, threatened, or voluntarily participated in a patent infringement lawsuit against Us or any of Our licensees.

4. DISCLAIMER THE CONTRIBUTION IS PROVIDED “AS IS” . MORE PARTICULARLY, ALL EXPRESS OR IMPLIED WARRANTIES INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT ARE EXPRESSLY DISCLAIMED BY YOU TO US AND BY US TO YOU. TO THE EXTENT THAT ANY SUCH WARRANTIES CANNOT BE DISCLAIMED, SUCH WARRANTY IS LIMITED IN DURATION TO THE MINIMUM PERIOD PERMITTED BY LAW.

5. Consequential Damage Waiver TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT WILL YOU OR US BE LIABLE FOR ANY LOSS OF PROFITS, LOSS OF ANTICIPATED SAVINGS, LOSS OF DATA, INDIRECT, SPECIAL, INCIDENTAL, CONSEQUENTIAL AND EXEMPLARY DAMAGES ARISING OUT OF THIS AGREEMENT REGARDLESS OF THE LEGAL OR EQUITABLE THEORY (CONTRACT, TORT OR OTHERWISE) UPON WHICH THE CLAIM IS BASED.

6. Approximation of Disclaimer and Damage Waiver IF THE DISCLAIMER AND DAMAGE WAIVER MENTIONED IN SECTION 4 AND SECTION 5 CANNOT BE GIVEN LEGAL EFFECT UNDER APPLICABLE LOCAL LAW, REVIEWING COURTS SHALL APPLY LOCAL LAW THAT MOST CLOSELY APPROXIMATES AN ABSOLUTE WAIVER OF ALL CIVIL LIABILITY IN CONNECTION WITH THE CONTRIBUTION.

7. Term 7.1 This Agreement shall come into effect upon Your acceptance of the terms and conditions.

7.2 In the event of a termination of this Agreement Sections 4, 5, 6, 7 and 8 shall survive such termination and shall remain in full force thereafter. For the avoidance of doubt, Contributions that are already licensed under a free and open source license at the date of the termination shall remain in full force after the termination of this Agreement.

8. Miscellaneous 8.1 This Agreement and all disputes, claims, actions, suits or other proceedings arising out of this agreement or relating in any way to it shall be governed by the laws of People’s Republic of China excluding its private international law provisions.

8.2 This Agreement sets out the entire agreement between You and Us for Your Contributions to Us and overrides all other agreements or understandings.

8.3 If any provision of this Agreement is found void and unenforceable, such provision will be replaced to the extent possible with a provision that comes closest to the meaning of the original provision and that is enforceable. The terms and conditions set forth in this Agreement shall apply notwithstanding any failure of essential purpose of this Agreement or any limited remedy to the maximum extent possible under law.

8.4 You agree to notify Us of any facts or circumstances of which you become aware that would make this Agreement inaccurate in any respect.

You

| | |
|----------|--|
| Date: | |
| Name: | |
| Title: | |
| Address: | |

Us

| | |
|----------|--|
| Date: | |
| Name: | |
| Title: | |
| Address: | |

Chapter 8

ESP-IDF Versions

The ESP-IDF GitHub repository is updated regularly, especially the master branch where new development takes place.

For production use, there are also stable releases available.

8.1 Releases

The documentation for the current stable release version can always be found at this URL:

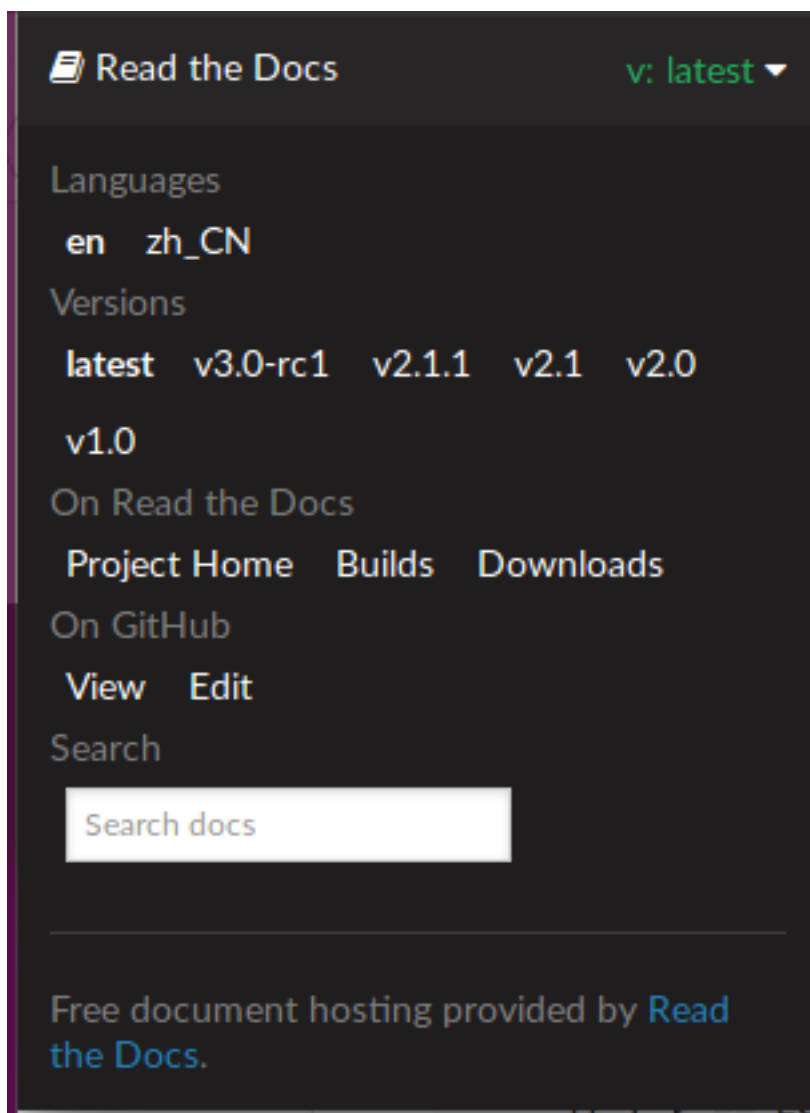
<https://docs.espressif.com/projects/esp-idf/en/stable/>

Documentation for the latest version (master branch) can always be found at this URL:

<https://docs.espressif.com/projects/esp-idf/en/latest/>

The full history of releases can be found on the GitHub repository [Releases page](#). There you can find release notes, links to each version of the documentation, and instructions for obtaining each version.

Another place to find documentation for all releases is the documentation page, where you can go to the bottom-left corner and click the versions dropdown (a bar with a small triangle). You can also use this dropdown to switch between versions of the documentation.



8.2 Which Version Should I Start With?

- For production purposes, use the [current stable version](#). Stable versions have been manually tested, and are updated with “bugfix releases” which fix bugs without changing other functionality (see [Versioning Scheme](#) for more details). Every stable release version can be found on the [Releases page](#).
- For prototyping, experimentation or for developing new ESP-IDF features, use the [latest version \(master branch in Git\)](#). The latest version in the master branch has all the latest features and has passed automated testing, but has not been completely manually tested (“bleeding edge”).
- If a required feature is not yet available in a stable release, but you do not want to use the master branch, it is possible to check out a pre-release version or a release branch. It is recommended to start from a stable version and then follow the instructions for [Updating to a Pre-Release Version](#) or [Updating to a Release Branch](#).

See [Updating ESP-IDF](#) if you already have a local copy of ESP-IDF and wish to update it.

8.3 Versioning Scheme

ESP-IDF uses [Semantic Versioning](#). This means that:

- Major Releases, like `v3.0`, add new functionality and may change functionality. This includes removing deprecated functionality.

If updating to a new major release (for example, from v2.1 to v3.0), some of your project's code may need updating and functionality may need to be re-tested. The release notes on the [Releases page](#) include lists of Breaking Changes to refer to.

- Minor Releases like v3.1 add new functionality and fix bugs but will not change or remove documented functionality, or make incompatible changes to public APIs.

If updating to a new minor release (for example, from v3.0 to v3.1), your project's code does not require updating, but you should re-test your project. Pay particular attention to the items mentioned in the release notes on the [Releases page](#).

- Bugfix Releases like v3.0.1 only fix bugs and do not add new functionality.

If updating to a new bugfix release (for example, from v3.0 to v3.0.1), you do not need to change any code in your project, and you only need to re-test the functionality directly related to bugs listed in the release notes on the [Releases page](#).

8.4 Support Periods

Each ESP-IDF major and minor release version has an associated support period. After this period, the release is End of Life and no longer supported.

The [ESP-IDF Support Period Policy](#) explains this in detail, and describes how the support periods for each release are determined.

Each release on the [Releases page](#) includes information about the support period for that particular release.

As a general guideline:

- If starting a new project, use the latest stable release.
- If you have a GitHub account, click the “Watch” button in the top-right of the [Releases page](#) and choose “Releases only”. GitHub will notify you whenever a new release is available. Whenever a bug fix release is available for the version you are using, plan to update to it.
- If possible, periodically update the project to a new major or minor ESP-IDF version (for example, once a year.) The update process should be straightforward for Minor updates, but may require some planning and checking of the release notes for Major updates.
- Always plan to update to a newer release before the release you are using becomes End of Life.

8.5 Checking the Current Version

The local ESP-IDF version can be checked by using `idf.py`:

```
idf.py --version
```

The ESP-IDF version is also compiled into the firmware and can be accessed (as a string) via the macro `IDF_VER`. The default ESP-IDF bootloader will print the version on boot (the version information is not always updated if the code in the GitHub repo is updated, it only changes if there is a clean build or if that particular source file is recompiled).

If writing code that needs to support multiple ESP-IDF versions, the version can be checked at compile time using [compile-time macros](#).

Examples of ESP-IDF versions:

| Version String | Meaning |
|--------------------------|--|
| v3.2-dev-306-gbeb3611ca | Master branch pre-release. - v3.2-dev - in development for version 3.2. - 306 - number of commits after v3.2 development started. - beb3611ca - commit identifier. |
| v3.0.2 | Stable release, tagged v3.0.2. |
| v3.1-beta1-75-g346d6b0ea | Beta version in development (on a <i>release branch</i>). - v3.1-beta1 - pre-release tag. - 75 - number of commits after the pre-release beta tag was assigned. - 346d6b0ea - commit identifier. |
| v3.0.1-dirty | Stable release, tagged v3.0.1. - dirty means that there are modifications in the local ESP-IDF directory. |

8.6 Git Workflow

The development (Git) workflow of the Espressif ESP-IDF team is as follows:

- New work is always added on the master branch (latest version) first. The ESP-IDF version on `master` is always tagged with `-dev` (for “in development”), for example `v3.1-dev`.
- Changes are first added to an internal Git repository for code review and testing but are pushed to GitHub after automated testing passes.
- When a new version (developed on `master`) becomes feature complete and “beta” quality, a new branch is made for the release, for example `release/v3.1`. A pre-release tag is also created, for example `v3.1-beta1`. You can see a full [list of branches](#) and a [list of tags](#) on GitHub. Beta pre-releases have release notes which may include a significant number of Known Issues.
- As testing of the beta version progresses, bug fixes will be added to both the `master` branch and the release branch. New features for the next release may start being added to `master` at the same time.
- Once testing is nearly complete a new release candidate is tagged on the release branch, for example `v3.1-rc1`. This is still a pre-release version.
- If no more significant bugs are found or reported, then the final Major or Minor Version is tagged, for example `v3.1`. This version appears on the [Releases page](#).
- As bugs are reported in released versions, the fixes will continue to be committed to the same release branch.
- Regular bugfix releases are made from the same release branch. After manual testing is complete, a bugfix release is tagged (i.e. `v3.1.1`) and appears on the [Releases page](#).

8.7 Updating ESP-IDF

Updating ESP-IDF depends on which version(s) you wish to follow:

- [Updating to Stable Release](#) is recommended for production use.

- [Updating to Master Branch](#) is recommended for the latest features, development use, and testing.
- [Updating to a Release Branch](#) is a compromise between the first two.

Note: These guides assume that you already have a local copy of ESP-IDF cloned. To get one, check Step 2 in the [Getting Started](#) guide for any ESP-IDF version.

8.7.1 Updating to Stable Release

To update to a new ESP-IDF release (recommended for production use), this is the process to follow:

- Check the [Releases page](#) regularly for new releases.
- When a bugfix release for the version you are using is released (for example, if using v3.0.1 and v3.0.2 is released), check out the new bugfix version into the existing ESP-IDF directory:

```
cd $IDF_PATH
git fetch
git checkout vX.Y.Z
git submodule update --init --recursive
```

- When major or minor updates are released, check the Release Notes on the releases page and decide if you want to update or to stay with your current release. Updating is via the same Git commands shown above.

Note: If you installed the stable release via zip file instead of using git, it might not be possible to update versions using the commands. In this case, update by downloading a new zip file and replacing the entire `IDF_PATH` directory with its contents.

8.7.2 Updating to a Pre-Release Version

It is also possible to `git checkout` a tag corresponding to a pre-release version or release candidate, the process is the same as [Updating to Stable Release](#).

Pre-release tags are not always found on the [Releases page](#). Consult the [list of tags](#) on GitHub for a full list. Caveats for using a pre-release are similar to [Updating to a Release Branch](#).

8.7.3 Updating to Master Branch

Note: Using Master branch means living “on the bleeding edge” with the latest ESP-IDF code.

To use the latest version on the ESP-IDF master branch, this is the process to follow:

- Check out the master branch locally:

```
cd $IDF_PATH
git checkout master
git pull
git submodule update --init --recursive
```

- Periodically, re-run `git pull` to pull the latest version of master. Note that you may need to change your project or report bugs after updating your master branch.
- To switch from master to a release branch or stable version, run `git checkout` as shown in the other sections.

Important: It is strongly recommended to regularly run `git pull` and then `git submodule update --init --recursive` so a local copy of master does not get too old. Arbitrary old master branch revisions are effectively unsupported “snapshots” that may have undocumented bugs. For a semi-stable version, try [Updating to a Release Branch](#) instead.

8.7.4 Updating to a Release Branch

In terms of stability, using a release branch is part-way between using the master branch and only using stable releases. A release branch is always beta quality or better, and receives bug fixes before they appear in each stable release.

You can find a [list of branches](#) on GitHub.

For example, to follow the branch for ESP-IDF v3.1, including any bugfixes for future releases like v3.1.1, etc:

```
cd $IDF_PATH
git fetch
git checkout release/v3.1
git pull
git submodule update --init --recursive
```

Each time you `git pull` this branch, ESP-IDF will be updated with fixes for this release.

Note: There is no dedicated documentation for release branches. It is recommended to use the documentation for the closest version to the branch which is currently checked out.

Chapter 9

Resources

- The [esp32.com forum](#) is a place to ask questions and find community resources.
- Check the [Issues](#) section on GitHub if you find a bug or have a feature request. Please check existing [Issues](#) before opening a new one.
- A comprehensive collection of [solutions, practical applications, components and drivers](#) based on ESP-IDF is available in [ESP32 IoT Solution](#) repository. In most of cases descriptions are provided both in English and in 中文.
- To develop applications using Arduino platform, refer to [Arduino core for ESP32 WiFi chip](#).
- Several [books](#) have been written about ESP32 and they are listed on [Espressif](#) web site.
- If you' re interested in contributing to ESP-IDF, please check the [Contributions Guide](#).
- For additional ESP32 product related information, please refer to [documentation](#) section of [Espressif](#) site.
- [Download](#) latest and previous versions of this documentation in PDF and HTML format.

Chapter 10

Copyrights and Licenses

10.1 Software Copyrights

All original source code in this repository is Copyright (C) 2015-2019 Espressif Systems. This source code is licensed under the Apache License 2.0 as described in the file LICENSE.

Additional third party copyrighted code is included under the following licenses.

Where source code headers specify Copyright & License information, this information takes precedence over the summaries made here.

10.1.1 Firmware Components

These third party libraries can be included into the application (firmware) produced by ESP-IDF.

- [Newlib](#) is licensed under the BSD License and is Copyright of various parties, as described in [COPYING.NEWLIB](#).
- [Xtensa header files](#) are Copyright (C) 2013 Tensilica Inc and are licensed under the MIT License as reproduced in the individual header files.
- Original parts of [FreeRTOS](#) (components/freertos) are Copyright (C) 2015 Real Time Engineers Ltd and is licensed under the GNU General Public License V2 with the FreeRTOS Linking Exception, as described in [license.txt](#).
- Original parts of [LWIP](#) (components/lwip) are Copyright (C) 2001, 2002 Swedish Institute of Computer Science and are licensed under the BSD License as described in [COPYING file](#).
- [wpa_supplicant](#) Copyright (c) 2003-2005 Jouni Malinen and licensed under the BSD license.
- [FreeBSD net80211](#) Copyright (c) 2004-2008 Sam Leffler, Errno Consulting and licensed under the BSD license.
- [JSMN](#) JSON Parser (components/jsmn) Copyright (c) 2010 Serge A. Zaitsev and licensed under the MIT license.
- [argtable3](#) argument parsing library Copyright (C) 1998-2001,2003-2011,2013 Stewart Heitmann and licensed under 3-clause BSD license.
- [linenoise](#) line editing library Copyright (c) 2010-2014 Salvatore Sanfilippo, Copyright (c) 2010-2013 Pieter Noordhuis, licensed under 2-clause BSD license.
- [libcoap](#) COAP library Copyright (c) 2010-2017 Olaf Bergmann and others, is licensed under 2-clause BSD license.
- [libexpat](#) XML parsing library Copyright (c) 1998-2000 Thai Open Source Software Center Ltd and Clark Cooper, Copyright (c) 2001-2017 Expat maintainers, is licensed under MIT license.
- [FatFS](#) library, Copyright (C) 2017 ChaN, is licensed under [a BSD-style license](#) .
- [cJSON](#) library, Copyright (c) 2009-2017 Dave Gamble and cJSON contributors, is licensed under MIT license.
- [libsodium](#) library, Copyright (c) 2013-2018 Frank Denis, is licensed under ISC license.
- [micro-ecc](#) library, Copyright (c) 2014 Kenneth MacKay, is licensed under 2-clause BSD license.
- [nghttp2](#) library, Copyright (c) 2012, 2014, 2015, 2016 Tatsuhiro Tsujikawa, Copyright (c) 2012, 2014, 2015, 2016 nghttp2 contributors, is licensed under MIT license.

- [Mbed TLS](#) library, Copyright (C) 2006-2018 ARM Limited, is licensed under Apache License 2.0.
- [SPIFFS](#) library, Copyright (c) 2013-2017 Peter Andersson, is licensed under MIT license.
- [TinyCBOR](#) library, Copyright (c) 2017 Intel Corporation, is licensed under MIT License.
- [SD/MMC driver](#) is derived from [OpenBSD SD/MMC driver](#), Copyright (c) 2006 Uwe Stuehler, and is licensed under BSD license.
- [Asio](#) , Copyright (c) 2003-2018 Christopher M. Kohlhoff is licensed under the Boost Software License.
- [ESP-MQTT](#) MQTT Package (contiki-mqtt) - Copyright (c) 2014, Stephen Robinson, MQTT-ESP - Tuan PM <tuanpm at live dot com> is licensed under Apache License 2.0.
- [BLE Mesh](#) is adapted from Zephyr Project, Copyright (c) 2017-2018 Intel Corporation and licensed under Apache License 2.0
- [mynewt-nimble](#) Apache Mynewt NimBLE, Copyright 2015-2018, The Apache Software Foundation, is licensed under Apache License 2.0.

10.1.2 Build Tools

This is the list of licenses for tools included in this repository, which are used to build applications. The tools do not become part of the application (firmware), so their license does not affect licensing of the application.

- [esptool.py](#) is Copyright (C) 2014-2016 Fredrik Ahlberg, Angus Gratton and is licensed under the GNU General Public License v2, as described in [LICENSE file](#).
- [KConfig](#) is Copyright (C) 2002 Roman Zippel and others, and is licensed under the GNU General Public License V2.
- [Kconfiglib](#) is Copyright (C) 2011-2019, Ulf Magnusson, and is licensed under the ISC License.
- [Menuconfig of Kconfiglib](#) is Copyright (C) 2018-2019, Nordic Semiconductor ASA and Ulf Magnusson, and is licensed under the ISC License.

10.2 ROM Source Code Copyrights

ESP32 mask ROM hardware includes binaries compiled from portions of the following third party software:

- [Newlib](#) , licensed under the BSD License and is Copyright of various parties, as described in [COPYING.NEWLIB](#).
- [Xtensa libhal](#), Copyright (c) Tensilica Inc and licensed under the MIT license (see below).
- [TinyBasic Plus](#), Copyright Mike Field & Scott Lawrence and licensed under the MIT license (see below).
- [miniz](#), by Rich Geldreich - placed into the public domain.
- [wpa_supplicant](#) Copyright (c) 2003-2005 Jouni Malinen and licensed under the BSD license.
- [TJpgDec](#) Copyright (C) 2011, ChaN, all right reserved. See below for license.

10.3 Xtensa libhal MIT License

Copyright (c) 2003, 2006, 2010 Tensilica Inc.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

10.4 TinyBasic Plus MIT License

Copyright (c) 2012-2013

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

10.5 TjpgDec License

TjpgDec - Tiny JPEG Decompressor R0.01 (C)ChaN, 2011 The TjpgDec is a generic JPEG decompressor module for tiny embedded systems. This is a free software that opened for education, research and commercial developments under license policy of following terms.

Copyright (C) 2011, ChaN, all right reserved.

- The TjpgDec module is a free software and there is NO WARRANTY.
- No restriction on use. You can use, modify and redistribute it for personal, non-profit or commercial products UNDER YOUR RESPONSIBILITY.
- Redistributions of source code must retain the above copyright notice.

Chapter 11

About

This is documentation of [ESP-IDF](#), the framework to develop applications for [ESP32](#) chip by [Espressif](#).

The ESP32 is 2.4 GHz Wi-Fi and Bluetooth combo, which integrates one or two 32-bit microprocessors, with up to 600 DMIPS processing power.

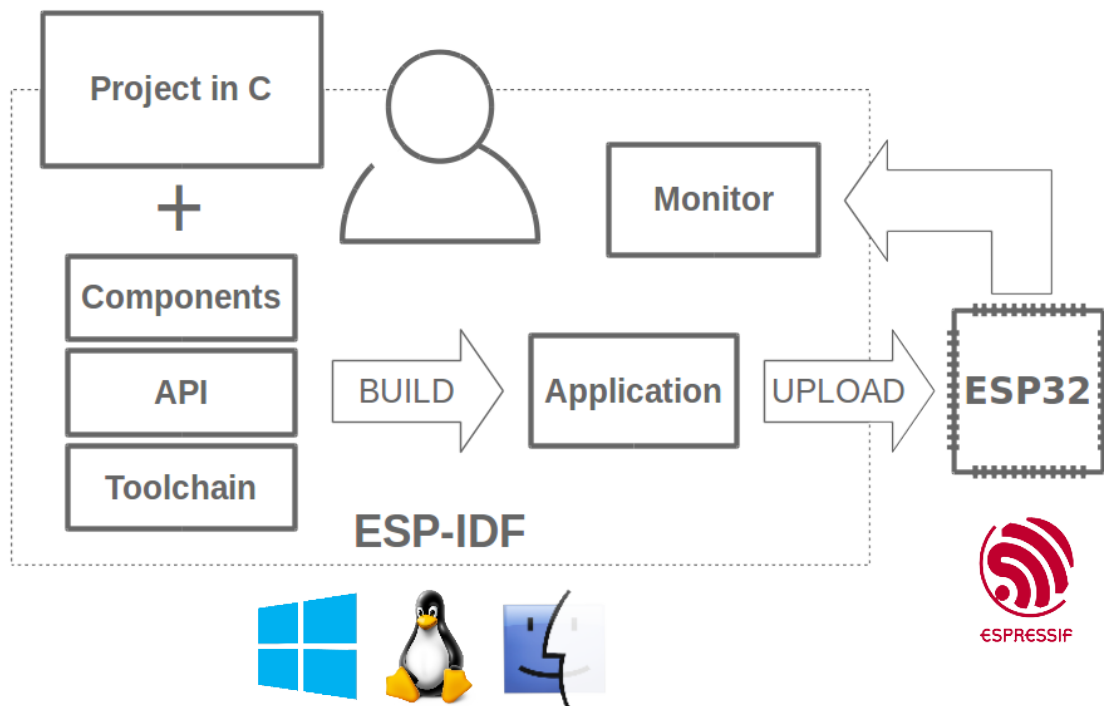


Fig. 1: Espressif IoT Integrated Development Framework

The ESP-IDF, Espressif IoT Development Framework, provides toolchain, API, components and workflows to develop applications for ESP32 using Windows, Linux and Mac OS operating systems.

Chapter 12

Switch Between Languages/切换语言

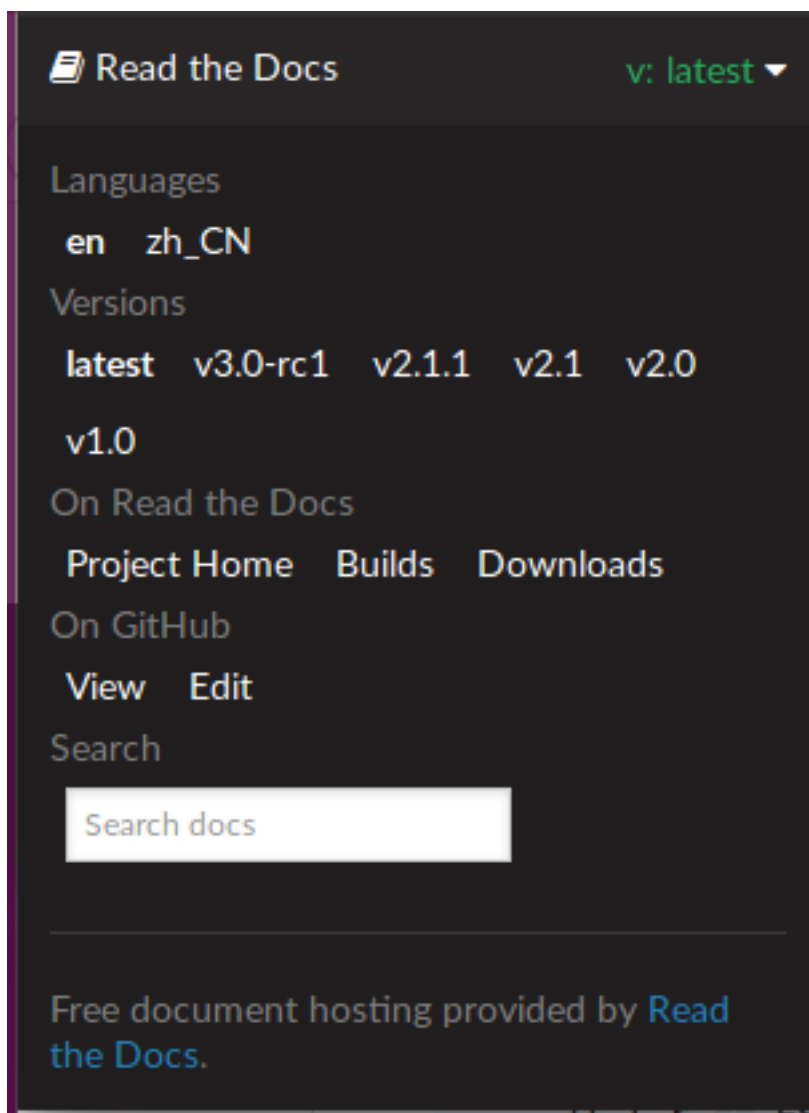
The ESP-IDF Programming Manual is now available in two languages. Please refer to the English version if there is any discrepancy.

《ESP-IDF 编程手册》部分文档现在有两种语言的版本。如有出入请以英文版本为准。

- English/英文
- Chinese/中文

You can easily change from one language to another by the panel on the sidebar like below. Just click on the **Read the Docs** title button on the left-bottom corner if it is folded.

如下图所示，你可使用边栏的面板进行语言的切换。如果该面板被折叠，点击左下角 **Read the Docs** 标题按钮来显示它。



- [genindex](#)

Index

Symbols

[anonymous] (C++ *enum*), 118, 329

A

- ADC1_CHANNEL_0 (C++ *enumerator*), 597
- ADC1_CHANNEL_0_GPIO_NUM (C *macro*), 601
- ADC1_CHANNEL_1 (C++ *enumerator*), 597
- ADC1_CHANNEL_1_GPIO_NUM (C *macro*), 601
- ADC1_CHANNEL_2 (C++ *enumerator*), 597
- ADC1_CHANNEL_2_GPIO_NUM (C *macro*), 601
- ADC1_CHANNEL_3 (C++ *enumerator*), 597
- ADC1_CHANNEL_3_GPIO_NUM (C *macro*), 601
- ADC1_CHANNEL_4 (C++ *enumerator*), 597
- ADC1_CHANNEL_4_GPIO_NUM (C *macro*), 602
- ADC1_CHANNEL_5 (C++ *enumerator*), 597
- ADC1_CHANNEL_5_GPIO_NUM (C *macro*), 602
- ADC1_CHANNEL_6 (C++ *enumerator*), 597
- ADC1_CHANNEL_6_GPIO_NUM (C *macro*), 602
- ADC1_CHANNEL_7 (C++ *enumerator*), 597
- ADC1_CHANNEL_7_GPIO_NUM (C *macro*), 602
- adc1_channel_t (C++ *enum*), 597
- adc1_config_channel_atten (C++ *function*), 593
- adc1_config_width (C++ *function*), 593
- adc1_get_raw (C++ *function*), 594
- ADC1_GPIO32_CHANNEL (C *macro*), 602
- ADC1_GPIO33_CHANNEL (C *macro*), 602
- ADC1_GPIO34_CHANNEL (C *macro*), 602
- ADC1_GPIO35_CHANNEL (C *macro*), 602
- ADC1_GPIO36_CHANNEL (C *macro*), 601
- ADC1_GPIO37_CHANNEL (C *macro*), 601
- ADC1_GPIO38_CHANNEL (C *macro*), 601
- ADC1_GPIO39_CHANNEL (C *macro*), 601
- adc1_pad_get_io_num (C++ *function*), 593
- adc1_ulp_enable (C++ *function*), 595
- ADC2_CHANNEL_0 (C++ *enumerator*), 597
- ADC2_CHANNEL_0_GPIO_NUM (C *macro*), 602
- ADC2_CHANNEL_1 (C++ *enumerator*), 597
- ADC2_CHANNEL_1_GPIO_NUM (C *macro*), 602
- ADC2_CHANNEL_2 (C++ *enumerator*), 597
- ADC2_CHANNEL_2_GPIO_NUM (C *macro*), 602
- ADC2_CHANNEL_3 (C++ *enumerator*), 597
- ADC2_CHANNEL_3_GPIO_NUM (C *macro*), 602
- ADC2_CHANNEL_4 (C++ *enumerator*), 597
- ADC2_CHANNEL_4_GPIO_NUM (C *macro*), 602
- ADC2_CHANNEL_5 (C++ *enumerator*), 597
- ADC2_CHANNEL_5_GPIO_NUM (C *macro*), 602
- ADC2_CHANNEL_6 (C++ *enumerator*), 597
- ADC2_CHANNEL_6_GPIO_NUM (C *macro*), 602
- ADC2_CHANNEL_7 (C++ *enumerator*), 597
- ADC2_CHANNEL_7_GPIO_NUM (C *macro*), 602
- ADC2_CHANNEL_8 (C++ *enumerator*), 597
- ADC2_CHANNEL_8_GPIO_NUM (C *macro*), 602
- ADC2_CHANNEL_9 (C++ *enumerator*), 597
- ADC2_CHANNEL_9_GPIO_NUM (C *macro*), 602
- ADC2_CHANNEL_MAX (C++ *enumerator*), 598
- adc2_channel_t (C++ *enum*), 597
- adc2_config_channel_atten (C++ *function*), 596
- adc2_get_raw (C++ *function*), 596
- ADC2_GPIO0_CHANNEL (C *macro*), 602
- ADC2_GPIO12_CHANNEL (C *macro*), 602
- ADC2_GPIO13_CHANNEL (C *macro*), 602
- ADC2_GPIO14_CHANNEL (C *macro*), 602
- ADC2_GPIO15_CHANNEL (C *macro*), 602
- ADC2_GPIO25_CHANNEL (C *macro*), 602
- ADC2_GPIO26_CHANNEL (C *macro*), 602
- ADC2_GPIO27_CHANNEL (C *macro*), 602
- ADC2_GPIO2_CHANNEL (C *macro*), 602
- ADC2_GPIO4_CHANNEL (C *macro*), 602
- adc2_pad_get_io_num (C++ *function*), 595
- adc2_vref_to_gpio (C++ *function*), 596
- ADC_ATTEN_0db (C *macro*), 596
- ADC_ATTEN_11db (C *macro*), 596
- ADC_ATTEN_2_5db (C *macro*), 596
- ADC_ATTEN_6db (C *macro*), 596
- ADC_ATTEN_DB_0 (C++ *enumerator*), 599
- ADC_ATTEN_DB_11 (C++ *enumerator*), 599
- ADC_ATTEN_DB_2_5 (C++ *enumerator*), 599
- ADC_ATTEN_DB_6 (C++ *enumerator*), 599
- ADC_ATTEN_MAX (C++ *enumerator*), 599
- adc_atten_t (C++ *enum*), 599
- adc_bits_width_t (C++ *enum*), 599
- ADC_CHANNEL_0 (C++ *enumerator*), 598
- ADC_CHANNEL_1 (C++ *enumerator*), 598
- ADC_CHANNEL_2 (C++ *enumerator*), 598
- ADC_CHANNEL_3 (C++ *enumerator*), 598
- ADC_CHANNEL_4 (C++ *enumerator*), 598
- ADC_CHANNEL_5 (C++ *enumerator*), 598
- ADC_CHANNEL_6 (C++ *enumerator*), 598
- ADC_CHANNEL_7 (C++ *enumerator*), 598
- ADC_CHANNEL_8 (C++ *enumerator*), 598
- ADC_CHANNEL_9 (C++ *enumerator*), 598
- ADC_CHANNEL_MAX (C++ *enumerator*), 599

[adc_channel_t \(C++ enum\), 598](#)
[ADC_ENCODE_11BIT \(C++ enumerator\), 598](#)
[ADC_ENCODE_12BIT \(C++ enumerator\), 598](#)
[ADC_ENCODE_MAX \(C++ enumerator\), 598](#)
[adc_gpio_init \(C++ function\), 594](#)
[ADC_I2S_DATA_SRC_ADC \(C++ enumerator\), 599](#)
[ADC_I2S_DATA_SRC_IO_SIG \(C++ enumerator\), 599](#)
[ADC_I2S_DATA_SRC_MAX \(C++ enumerator\), 599](#)
[adc_i2s_encode_t \(C++ enum\), 598](#)
[adc_i2s_mode_init \(C++ function\), 595](#)
[adc_i2s_source_t \(C++ enum\), 599](#)
[adc_power_acquire \(C++ function\), 594](#)
[adc_power_off \(C++ function\), 594](#)
[adc_power_on \(C++ function\), 594](#)
[adc_power_release \(C++ function\), 594](#)
[adc_set_clk_div \(C++ function\), 595](#)
[adc_set_data_inv \(C++ function\), 594](#)
[adc_set_data_width \(C++ function\), 593](#)
[adc_set_i2s_data_source \(C++ function\), 595](#)
[ADC_UNIT_1 \(C++ enumerator\), 598](#)
[ADC_UNIT_2 \(C++ enumerator\), 598](#)
[ADC_UNIT_ALTER \(C++ enumerator\), 598](#)
[ADC_UNIT_BOTH \(C++ enumerator\), 598](#)
[ADC_UNIT_MAX \(C++ enumerator\), 598](#)
[adc_unit_t \(C++ enum\), 598](#)
[ADC_WIDTH_10Bit \(C macro\), 597](#)
[ADC_WIDTH_11Bit \(C macro\), 597](#)
[ADC_WIDTH_12Bit \(C macro\), 597](#)
[ADC_WIDTH_9Bit \(C macro\), 597](#)
[ADC_WIDTH_BIT_10 \(C++ enumerator\), 599](#)
[ADC_WIDTH_BIT_11 \(C++ enumerator\), 599](#)
[ADC_WIDTH_BIT_12 \(C++ enumerator\), 599](#)
[ADC_WIDTH_BIT_9 \(C++ enumerator\), 599](#)
[ADC_WIDTH_MAX \(C++ enumerator\), 599](#)
[ADD_DEV_FLUSHABLE_DEV_FLAG \(C macro\), 311](#)
[ADD_DEV_RM_AFTER_PROV_FLAG \(C macro\), 311](#)
[ADD_DEV_START_PROV_NOW_FLAG \(C macro\), 311](#)
[ADV_CHNL_37 \(C++ enumerator\), 148](#)
[ADV_CHNL_38 \(C++ enumerator\), 148](#)
[ADV_CHNL_39 \(C++ enumerator\), 149](#)
[ADV_CHNL_ALL \(C++ enumerator\), 149](#)
[ADV_FILTER_ALLOW_SCAN_ANY_CON_ANY \(C++ enumerator\), 149](#)
[ADV_FILTER_ALLOW_SCAN_ANY_CON_WLST \(C++ enumerator\), 149](#)
[ADV_FILTER_ALLOW_SCAN_WLST_CON_ANY \(C++ enumerator\), 149](#)
[ADV_FILTER_ALLOW_SCAN_WLST_CON_WLST \(C++ enumerator\), 149](#)
[ADV_TYPE_DIRECT_IND_HIGH \(C++ enumerator\), 148](#)
[ADV_TYPE_DIRECT_IND_LOW \(C++ enumerator\), 148](#)
[ADV_TYPE_IND \(C++ enumerator\), 148](#)
[ADV_TYPE_NONCONN_IND \(C++ enumerator\), 148](#)
[ADV_TYPE_SCAN_IND \(C++ enumerator\), 148](#)

B

[BD_ADDR \(C++ type\), 323](#)
[BD_ADDR_LEN \(C macro\), 311](#)
[BLE_ADDR_TYPE_PUBLIC \(C++ enumerator\), 123](#)
[BLE_ADDR_TYPE_RANDOM \(C++ enumerator\), 123](#)
[BLE_ADDR_TYPE_RPA_PUBLIC \(C++ enumerator\), 123](#)
[BLE_ADDR_TYPE_RPA_RANDOM \(C++ enumerator\), 123](#)
[BLE_BIT \(C macro\), 145](#)
[BLE_HCI_UART_H4_ACL \(C macro\), 277](#)
[BLE_HCI_UART_H4_CMD \(C macro\), 277](#)
[BLE_HCI_UART_H4_EVT \(C macro\), 277](#)
[BLE_HCI_UART_H4_NONE \(C macro\), 277](#)
[BLE_HCI_UART_H4_SCO \(C macro\), 277](#)
[BLE_SCAN_DUPLICATE_DISABLE \(C++ enumerator\), 150](#)
[BLE_SCAN_DUPLICATE_ENABLE \(C++ enumerator\), 150](#)
[BLE_SCAN_DUPLICATE_MAX \(C++ enumerator\), 150](#)
[BLE_SCAN_FILTER_ALLOW_ALL \(C++ enumerator\), 149](#)
[BLE_SCAN_FILTER_ALLOW_ONLY_WLST \(C++ enumerator\), 150](#)
[BLE_SCAN_FILTER_ALLOW_UND_RPA_DIR \(C++ enumerator\), 150](#)
[BLE_SCAN_FILTER_ALLOW_WLIST_PRA_DIR \(C++ enumerator\), 150](#)
[BLE_SCAN_TYPE_ACTIVE \(C++ enumerator\), 149](#)
[BLE_SCAN_TYPE_PASSIVE \(C++ enumerator\), 149](#)
[BLE_UUID128_VAL_LENGTH \(C macro\), 899](#)
[BLE_WL_ADDR_TYPE_PUBLIC \(C++ enumerator\), 123](#)
[BLE_WL_ADDR_TYPE_RANDOM \(C++ enumerator\), 123](#)
[BT_CONTROLLER_INIT_CONFIG_DEFAULT \(C macro\), 118](#)
[BT_OCTET32 \(C++ type\), 323](#)
[BT_OCTET32_LEN \(C macro\), 311](#)
[BTC_HF_CALL_END_TIMEOUT \(C macro\), 246](#)
[BTC_HF_INVALID_IDX \(C macro\), 246](#)
[BTC_HF_SECURITY \(C macro\), 246](#)
[BTC_HF_SERVICE_NAMES \(C macro\), 246](#)
[BTC_HF_SERVICES \(C macro\), 246](#)
[BTC_HFAG_SERVICE_NAME \(C macro\), 246](#)
[BTC_HSAG_SERVICE_NAME \(C macro\), 246](#)

C

[can_clear_receive_queue \(C++ function\), 616](#)
[can_clear_transmit_queue \(C++ function\), 615](#)
[can_driver_install \(C++ function\), 613](#)
[can_driver_uninstall \(C++ function\), 613](#)
[CAN_ERR_PASS_THRESH \(C macro\), 613](#)
[CAN_EXTD_ID_MASK \(C macro\), 612](#)
[can_filter_config_t \(C++ class\), 612](#)

- can_filter_config_t::acceptance_code (C++ member), 612
 can_filter_config_t::acceptance_mask (C++ member), 612
 can_filter_config_t::single_filter (C++ member), 612
 CAN_FRAME_EXTD_ID_LEN_BYTES (C macro), 613
 CAN_FRAME_MAX_DLC (C macro), 613
 CAN_FRAME_STD_ID_LEN_BYTES (C macro), 613
 can_general_config_t (C++ class), 616
 can_general_config_t::alerts_enabled (C++ member), 616
 can_general_config_t::bus_off_io (C++ member), 616
 can_general_config_t::clkout_divider (C++ member), 616
 can_general_config_t::clkout_io (C++ member), 616
 can_general_config_t::intr_flags (C++ member), 616
 can_general_config_t::mode (C++ member), 616
 can_general_config_t::rx_io (C++ member), 616
 can_general_config_t::rx_queue_len (C++ member), 616
 can_general_config_t::tx_io (C++ member), 616
 can_general_config_t::tx_queue_len (C++ member), 616
 can_get_status_info (C++ function), 615
 can_initiate_recovery (C++ function), 615
 CAN_IO_UNUSED (C macro), 617
 can_message_t (C++ class), 611
 can_message_t::data (C++ member), 612
 can_message_t::data_length_code (C++ member), 612
 can_message_t::dlc_non_comp (C++ member), 612
 can_message_t::extd (C++ member), 611
 can_message_t::flags (C++ member), 612
 can_message_t::identifier (C++ member), 612
 can_message_t::reserved (C++ member), 612
 can_message_t::rtr (C++ member), 611
 can_message_t::self (C++ member), 611
 can_message_t::ss (C++ member), 611
 CAN_MODE_LISTEN_ONLY (C++ enumerator), 613
 CAN_MODE_NO_ACK (C++ enumerator), 613
 CAN_MODE_NORMAL (C++ enumerator), 613
 can_mode_t (C++ enum), 613
 can_read_alerts (C++ function), 615
 can_receive (C++ function), 614
 can_reconfigure_alerts (C++ function), 615
 can_start (C++ function), 614
 CAN_STATE_BUS_OFF (C++ enumerator), 617
 CAN_STATE_RECOVERING (C++ enumerator), 617
 CAN_STATE_RUNNING (C++ enumerator), 617
 CAN_STATE_STOPPED (C++ enumerator), 617
 can_state_t (C++ enum), 617
 can_status_info_t (C++ class), 616
 can_status_info_t::arb_lost_count (C++ member), 617
 can_status_info_t::bus_error_count (C++ member), 617
 can_status_info_t::msgs_to_rx (C++ member), 616
 can_status_info_t::msgs_to_tx (C++ member), 616
 can_status_info_t::rx_error_counter (C++ member), 617
 can_status_info_t::rx_missed_count (C++ member), 617
 can_status_info_t::state (C++ member), 616
 can_status_info_t::tx_error_counter (C++ member), 617
 can_status_info_t::tx_failed_count (C++ member), 617
 CAN_STD_ID_MASK (C macro), 612
 can_stop (C++ function), 614
 can_timing_config_t (C++ class), 612
 can_timing_config_t::brp (C++ member), 612
 can_timing_config_t::sjw (C++ member), 612
 can_timing_config_t::triple_sampling (C++ member), 612
 can_timing_config_t::tseg_1 (C++ member), 612
 can_timing_config_t::tseg_2 (C++ member), 612
 can_transmit (C++ function), 614
 CHIP_ESP32 (C++ enumerator), 1177
 CHIP_ESP32S2BETA (C++ enumerator), 1177
 CHIP_FEATURE_BLE (C macro), 1176
 CHIP_FEATURE_BT (C macro), 1176
 CHIP_FEATURE_EMB_FLASH (C macro), 1176
 CHIP_FEATURE_WIFI_BGN (C macro), 1176
 CONFIG_EFUSE_CUSTOM_TABLE, 1008
 CONFIG_EFUSE_MAX_BLK_LEN, 1008
 CONFIG_EFUSE_VIRTUAL, 1011
 CONFIG_ESPTOOLPY_FLASHSIZE, 963
 CONFIG_FEATURE_WPA3_SAE_BIT (C macro), 503
 CONFIG_HEAP_TRACING_STACK_DEPTH (C macro), 1152
 CONFIG_LOG_DEFAULT_LEVEL, 1166, 1167
 CONFIG_LWIP_SNTP_UPDATE_DELAY, 1208
 CONFIG_LWIP_USE_ONLY_LWIP_SELECT, 987
 CONFIG_SPIRAM_BANKSWITCH_ENABLE, 1156
 CONFIG_SPIRAM_BANKSWITCH_RESERVE, 1156
- ## D
- DAC_CHANNEL_1 (C++ enumerator), 620

DAC_CHANNEL_1_GPIO_NUM (*C macro*), 619
 DAC_CHANNEL_2 (*C++ enumerator*), 620
 DAC_CHANNEL_2_GPIO_NUM (*C macro*), 619
 DAC_CHANNEL_MAX (*C++ enumerator*), 620
 dac_channel_t (*C++ enum*), 620
 dac_cw_config_t (*C++ class*), 619
 dac_cw_config_t::en_ch (*C++ member*), 619
 dac_cw_config_t::freq (*C++ member*), 620
 dac_cw_config_t::offset (*C++ member*), 620
 dac_cw_config_t::phase (*C++ member*), 619
 dac_cw_config_t::scale (*C++ member*), 619
 dac_cw_generator_config (*C++ function*), 619
 dac_cw_generator_disable (*C++ function*), 619
 dac_cw_generator_enable (*C++ function*), 619
 DAC_CW_PHASE_0 (*C++ enumerator*), 620
 DAC_CW_PHASE_180 (*C++ enumerator*), 620
 dac_cw_phase_t (*C++ enum*), 620
 DAC_CW_SCALE_1 (*C++ enumerator*), 620
 DAC_CW_SCALE_2 (*C++ enumerator*), 620
 DAC_CW_SCALE_4 (*C++ enumerator*), 620
 DAC_CW_SCALE_8 (*C++ enumerator*), 620
 dac_cw_scale_t (*C++ enum*), 620
 DAC_GPIO25_CHANNEL (*C macro*), 619
 DAC_GPIO26_CHANNEL (*C macro*), 619
 dac_i2s_disable (*C++ function*), 618
 dac_i2s_enable (*C++ function*), 618
 dac_output_disable (*C++ function*), 618
 dac_output_enable (*C++ function*), 618
 dac_output_voltage (*C++ function*), 618
 dac_pad_get_io_num (*C++ function*), 618
 DEFAULT_HTTP_BUF_SIZE (*C macro*), 829
 DEL_DEV_ADDR_FLAG (*C macro*), 311
 DEL_DEV_UUID_FLAG (*C macro*), 311

E

eAbortSleep (*C++ enumerator*), 1059
 eBlocked (*C++ enumerator*), 1058
 eDeleted (*C++ enumerator*), 1058
 EFUSE_CODE_SCHEME_SELECTOR, 1009
 eIncrement (*C++ enumerator*), 1059
 eNoAction (*C++ enumerator*), 1058
 eNoTasksWaitingTimeout (*C++ enumerator*), 1059
 eNotifyAction (*C++ enum*), 1058
 environment variable
 CONFIG_EFUSE_CUSTOM_TABLE, 1008
 CONFIG_EFUSE_MAX_BLK_LEN, 1008
 CONFIG_EFUSE_VIRTUAL, 1011
 CONFIG_ESPTOOLPY_FLASHSIZE, 963
 CONFIG_LOG_DEFAULT_LEVEL, 1166, 1167
 CONFIG_LWIP_SNTP_UPDATE_DELAY, 1208
 CONFIG_LWIP_USE_ONLY_LWIP_SELECT, 987
 CONFIG_SPIRAM_BANKSWITCH_ENABLE, 1156
 CONFIG_SPIRAM_BANKSWITCH_RESERVE, 1156

EFUSE_CODE_SCHEME_SELECTOR, 1009
 eReady (*C++ enumerator*), 1058
 eRunning (*C++ enumerator*), 1058
 eSetBits (*C++ enumerator*), 1059
 eSetValueWithoutOverwrite (*C++ enumerator*), 1059
 eSetValueWithOverwrite (*C++ enumerator*), 1059
 eSleepModeStatus (*C++ enum*), 1059
 ESP_A2D_AUDIO_CFG_EVT (*C++ enumerator*), 221
 ESP_A2D_AUDIO_STATE_EVT (*C++ enumerator*), 221
 ESP_A2D_AUDIO_STATE_REMOTE_SUSPEND (*C++ enumerator*), 220
 ESP_A2D_AUDIO_STATE_STARTED (*C++ enumerator*), 220
 ESP_A2D_AUDIO_STATE_STOPPED (*C++ enumerator*), 220
 esp_a2d_audio_state_t (*C++ enum*), 220
 esp_a2d_cb_event_t (*C++ enum*), 221
 esp_a2d_cb_param_t (*C++ union*), 218
 esp_a2d_cb_param_t::a2d_audio_cfg_param (*C++ class*), 218
 esp_a2d_cb_param_t::a2d_audio_cfg_param::mcc (*C++ member*), 218
 esp_a2d_cb_param_t::a2d_audio_cfg_param::remote_b (*C++ member*), 218
 esp_a2d_cb_param_t::a2d_audio_stat_param (*C++ class*), 218
 esp_a2d_cb_param_t::a2d_audio_stat_param::remote_b (*C++ member*), 218
 esp_a2d_cb_param_t::a2d_audio_stat_param::state (*C++ member*), 218
 esp_a2d_cb_param_t::a2d_conn_stat_param (*C++ class*), 218
 esp_a2d_cb_param_t::a2d_conn_stat_param::disc_rsn (*C++ member*), 218
 esp_a2d_cb_param_t::a2d_conn_stat_param::remote_b (*C++ member*), 218
 esp_a2d_cb_param_t::a2d_conn_stat_param::state (*C++ member*), 218
 esp_a2d_cb_param_t::a2d_prof_stat (*C++ member*), 218
 esp_a2d_cb_param_t::a2d_prof_stat_param (*C++ class*), 218
 esp_a2d_cb_param_t::a2d_prof_stat_param::init_sta (*C++ member*), 219
 esp_a2d_cb_param_t::audio_cfg (*C++ member*), 218
 esp_a2d_cb_param_t::audio_stat (*C++ member*), 218
 esp_a2d_cb_param_t::conn_stat (*C++ member*), 218
 esp_a2d_cb_param_t::media_ctrl_stat (*C++ member*), 218
 esp_a2d_cb_param_t::media_ctrl_stat_param (*C++ class*), 219
 esp_a2d_cb_param_t::media_ctrl_stat_param::cmd

- (C++ member), 219
- esp_a2d_cb_param_t::media_ctrl_stat_param (C++ member), 219
- esp_a2d_cb_t (C++ type), 219
- ESP_A2D_CIE_LEN_ATRAC (C macro), 219
- ESP_A2D_CIE_LEN_M12 (C macro), 219
- ESP_A2D_CIE_LEN_M24 (C macro), 219
- ESP_A2D_CIE_LEN_SBC (C macro), 219
- ESP_A2D_CONNECTION_STATE_CONNECTED (C++ enumerator), 220
- ESP_A2D_CONNECTION_STATE_CONNECTING (C++ enumerator), 220
- ESP_A2D_CONNECTION_STATE_DISCONNECTED (C++ enumerator), 220
- ESP_A2D_CONNECTION_STATE_DISCONNECTING (C++ enumerator), 220
- ESP_A2D_CONNECTION_STATE_EVT (C++ enumerator), 221
- esp_a2d_connection_state_t (C++ enum), 220
- ESP_A2D_DEINIT_SUCCESS (C++ enumerator), 221
- ESP_A2D_DISC_RSN_ABNORMAL (C++ enumerator), 220
- ESP_A2D_DISC_RSN_NORMAL (C++ enumerator), 220
- esp_a2d_disc_rsn_t (C++ enum), 220
- esp_a2d_init_state_t (C++ enum), 221
- ESP_A2D_INIT_SUCCESS (C++ enumerator), 221
- esp_a2d_mcc_t (C++ class), 219
- esp_a2d_mcc_t::cie (C++ member), 219
- esp_a2d_mcc_t::type (C++ member), 219
- ESP_A2D_MCT_ATRAC (C macro), 219
- ESP_A2D_MCT_M12 (C macro), 219
- ESP_A2D_MCT_M24 (C macro), 219
- ESP_A2D_MCT_NON_A2DP (C macro), 219
- ESP_A2D_MCT_SBC (C macro), 219
- esp_a2d_mct_t (C++ type), 219
- esp_a2d_media_ctrl (C++ function), 217
- ESP_A2D_MEDIA_CTRL_ACK_BUSY (C++ enumerator), 221
- ESP_A2D_MEDIA_CTRL_ACK_EVT (C++ enumerator), 221
- ESP_A2D_MEDIA_CTRL_ACK_FAILURE (C++ enumerator), 220
- ESP_A2D_MEDIA_CTRL_ACK_SUCCESS (C++ enumerator), 220
- esp_a2d_media_ctrl_ack_t (C++ enum), 220
- ESP_A2D_MEDIA_CTRL_CHECK_SRC_RDY (C++ enumerator), 221
- ESP_A2D_MEDIA_CTRL_NONE (C++ enumerator), 221
- ESP_A2D_MEDIA_CTRL_START (C++ enumerator), 221
- ESP_A2D_MEDIA_CTRL_STOP (C++ enumerator), 221
- ESP_A2D_MEDIA_CTRL_SUSPEND (C++ enumerator), 221
- esp_a2d_media_ctrl_t (C++ enum), 221
- ESP_A2D_PROF_STATE_EVT (C++ enumerator), 221
- esp_a2d_register_callback (C++ function), 216
- esp_a2d_sink_connect (C++ function), 216
- esp_a2d_sink_data_cb_t (C++ type), 219
- esp_a2d_sink_deinit (C++ function), 216
- esp_a2d_sink_disconnect (C++ function), 216
- esp_a2d_sink_init (C++ function), 216
- esp_a2d_sink_register_data_callback (C++ function), 216
- esp_a2d_source_connect (C++ function), 217
- esp_a2d_source_data_cb_t (C++ type), 220
- esp_a2d_source_deinit (C++ function), 217
- esp_a2d_source_disconnect (C++ function), 217
- esp_a2d_source_init (C++ function), 217
- esp_a2d_source_register_data_callback (C++ function), 217
- esp_adc_cal_characteristics_t (C++ class), 600
- esp_adc_cal_characteristics_t::adc_num (C++ member), 601
- esp_adc_cal_characteristics_t::atten (C++ member), 601
- esp_adc_cal_characteristics_t::bit_width (C++ member), 601
- esp_adc_cal_characteristics_t::coeff_a (C++ member), 601
- esp_adc_cal_characteristics_t::coeff_b (C++ member), 601
- esp_adc_cal_characteristics_t::high_curve (C++ member), 601
- esp_adc_cal_characteristics_t::low_curve (C++ member), 601
- esp_adc_cal_characteristics_t::vref (C++ member), 601
- esp_adc_cal_characterize (C++ function), 600
- esp_adc_cal_check_efuse (C++ function), 599
- esp_adc_cal_get_voltage (C++ function), 600
- esp_adc_cal_raw_to_voltage (C++ function), 600
- ESP_ADC_CAL_VAL_DEFAULT_VREF (C++ enumerator), 601
- ESP_ADC_CAL_VAL_EFUSE_TP (C++ enumerator), 601
- ESP_ADC_CAL_VAL_EFUSE_VREF (C++ enumerator), 601
- esp_adc_cal_value_t (C++ enum), 601
- ESP_APP_DESC_MAGIC_WORD (C macro), 1001
- esp_app_desc_t (C++ class), 1001
- esp_app_desc_t::app_elf_sha256 (C++ member), 1001
- esp_app_desc_t::date (C++ member), 1001
- esp_app_desc_t::idf_ver (C++ member), 1001

- esp_app_desc_t::magic_word (C++ member), 1001
 esp_app_desc_t::project_name (C++ member), 1001
 esp_app_desc_t::reserv1 (C++ member), 1001
 esp_app_desc_t::reserv2 (C++ member), 1001
 esp_app_desc_t::secure_version (C++ member), 1001
 esp_app_desc_t::time (C++ member), 1001
 esp_app_desc_t::version (C++ member), 1001
 ESP_APP_ID_MAX (C macro), 121
 ESP_APP_ID_MIN (C macro), 121
 esp_appttrace_buffer_get (C++ function), 1003
 esp_appttrace_buffer_put (C++ function), 1003
 esp_appttrace_dest_t (C++ enum), 1006
 ESP_APPTRACE_DEST_TRAX (C++ enumerator), 1006
 ESP_APPTRACE_DEST_UART0 (C++ enumerator), 1006
 esp_appttrace_down_buffer_config (C++ function), 1003
 esp_appttrace_down_buffer_get (C++ function), 1004
 esp_appttrace_down_buffer_put (C++ function), 1004
 esp_appttrace_fclose (C++ function), 1005
 esp_appttrace_flush (C++ function), 1004
 esp_appttrace_flush_nolock (C++ function), 1004
 esp_appttrace_fopen (C++ function), 1005
 esp_appttrace_fread (C++ function), 1005
 esp_appttrace_fseek (C++ function), 1005
 esp_appttrace_fstop (C++ function), 1006
 esp_appttrace_ftell (C++ function), 1006
 esp_appttrace_fwrite (C++ function), 1005
 esp_appttrace_host_is_connected (C++ function), 1005
 esp_appttrace_init (C++ function), 1003
 esp_appttrace_read (C++ function), 1004
 esp_appttrace_vprintf (C++ function), 1004
 esp_appttrace_vprintf_to (C++ function), 1004
 esp_appttrace_write (C++ function), 1003
 esp_attr_control_t (C++ class), 153
 esp_attr_control_t::auto_rsp (C++ member), 153
 esp_attr_desc_t (C++ class), 153
 esp_attr_desc_t::length (C++ member), 153
 esp_attr_desc_t::max_length (C++ member), 153
 esp_attr_desc_t::perm (C++ member), 153
 esp_attr_desc_t::uuid_length (C++ member), 153
 esp_attr_desc_t::uuid_p (C++ member), 153
 esp_attr_desc_t::value (C++ member), 153
 esp_attr_value_t (C++ class), 153
 esp_attr_value_t::attr_len (C++ member), 153
 esp_attr_value_t::attr_max_len (C++ member), 153
 esp_attr_value_t::attr_value (C++ member), 154
 ESP_AVRC_BATT_CRITICAL (C++ enumerator), 238
 ESP_AVRC_BATT_EXTERNAL (C++ enumerator), 238
 ESP_AVRC_BATT_FULL_CHARGE (C++ enumerator), 238
 ESP_AVRC_BATT_NORMAL (C++ enumerator), 237
 esp_avrc_batt_stat_t (C++ enum), 237
 ESP_AVRC_BATT_WARNING (C++ enumerator), 237
 ESP_AVRC_BIT_MASK_OP_CLEAR (C++ enumerator), 234
 ESP_AVRC_BIT_MASK_OP_SET (C++ enumerator), 234
 esp_avrc_bit_mask_op_t (C++ enum), 234
 ESP_AVRC_BIT_MASK_OP_TEST (C++ enumerator), 234
 esp_avrc_ct_cb_event_t (C++ enum), 234
 esp_avrc_ct_cb_param_t (C++ union), 226
 esp_avrc_ct_cb_param_t::avrc_ct_change_notify_param (C++ class), 226
 esp_avrc_ct_cb_param_t::avrc_ct_change_notify_param (C++ member), 226
 esp_avrc_ct_cb_param_t::avrc_ct_change_notify_param (C++ member), 226
 esp_avrc_ct_cb_param_t::avrc_ct_conn_stat_param (C++ class), 226
 esp_avrc_ct_cb_param_t::avrc_ct_conn_stat_param (C++ member), 227
 esp_avrc_ct_cb_param_t::avrc_ct_conn_stat_param (C++ member), 227
 esp_avrc_ct_cb_param_t::avrc_ct_get_rn_caps_rsp_param (C++ class), 227
 esp_avrc_ct_cb_param_t::avrc_ct_get_rn_caps_rsp_param (C++ member), 227
 esp_avrc_ct_cb_param_t::avrc_ct_get_rn_caps_rsp_param (C++ member), 227
 esp_avrc_ct_cb_param_t::avrc_ct_get_rn_caps_rsp_param (C++ member), 227
 esp_avrc_ct_cb_param_t::avrc_ct_meta_rsp_param (C++ class), 227
 esp_avrc_ct_cb_param_t::avrc_ct_meta_rsp_param (C++ member), 227
 esp_avrc_ct_cb_param_t::avrc_ct_meta_rsp_param (C++ member), 227
 esp_avrc_ct_cb_param_t::avrc_ct_meta_rsp_param (C++ member), 227
 esp_avrc_ct_cb_param_t::avrc_ct_psth_rsp_param (C++ class), 227
 esp_avrc_ct_cb_param_t::avrc_ct_psth_rsp_param (C++ member), 227
 esp_avrc_ct_cb_param_t::avrc_ct_psth_rsp_param (C++ member), 227

- (C++ member), 227
- esp_avrc_ct_cb_param_t::avrc_ct_psth_rsp_param (C++ function), 222
- (C++ member), 227
- esp_avrc_ct_cb_param_t::avrc_ct_rmt_feats_param (C++ member), 227
- (C++ class), 227
- esp_avrc_ct_cb_param_t::avrc_ct_rmt_feats_param::feat_mask (C++ member), 227
- esp_avrc_ct_cb_param_t::avrc_ct_rmt_feats_param::feat_mask (C++ member), 228
- esp_avrc_ct_cb_param_t::avrc_ct_rmt_feats_param::feat_mask (C++ member), 227
- esp_avrc_ct_cb_param_t::avrc_ct_set_volume_rsp_param (C++ class), 228
- esp_avrc_ct_cb_param_t::avrc_ct_set_volume_rsp_param (C++ member), 228
- esp_avrc_ct_cb_param_t::change_ntf (C++ member), 226
- esp_avrc_ct_cb_param_t::conn_stat (C++ member), 226
- esp_avrc_ct_cb_param_t::get_rn_caps_rsp (C++ member), 226
- esp_avrc_ct_cb_param_t::meta_rsp (C++ member), 226
- esp_avrc_ct_cb_param_t::psth_rsp (C++ member), 226
- esp_avrc_ct_cb_param_t::rmt_feats (C++ member), 226
- esp_avrc_ct_cb_param_t::set_volume_rsp (C++ member), 226
- esp_avrc_ct_cb_t (C++ type), 230
- ESP_AVRC_CT_CHANGE_NOTIFY_EVT (C++ enumerator), 234
- ESP_AVRC_CT_CONNECTION_STATE_EVT (C++ enumerator), 234
- esp_avrc_ct_deinit (C++ function), 222
- ESP_AVRC_CT_GET_RN_CAPABILITIES_RSP_EVT (C++ enumerator), 234
- esp_avrc_ct_init (C++ function), 222
- ESP_AVRC_CT_METADATA_RSP_EVT (C++ enumerator), 234
- ESP_AVRC_CT_PASSTHROUGH_RSP_EVT (C++ enumerator), 234
- ESP_AVRC_CT_PLAY_STATUS_RSP_EVT (C++ enumerator), 234
- esp_avrc_ct_register_callback (C++ function), 222
- ESP_AVRC_CT_REMOTE_FEATURES_EVT (C++ enumerator), 234
- esp_avrc_ct_send_get_rn_capabilities_cmd (C++ function), 222
- esp_avrc_ct_send_metadata_cmd (C++ function), 223
- esp_avrc_ct_send_passthrough_cmd (C++ function), 223
- esp_avrc_ct_send_register_notification_cmd (C++ function), 222
- esp_avrc_ct_send_set_absolute_volume_cmd (C++ function), 223
- esp_avrc_ct_send_set_player_value_cmd (C++ function), 222
- ESP_AVRC_CT_SET_ABSOLUTE_VOLUME_RSP_EVT (C++ enumerator), 234
- ESP_AVRC_FEAT_ADV_CTRL (C++ enumerator), 230
- ESP_AVRC_FEAT_BROWSE (C++ enumerator), 230
- ESP_AVRC_FEAT_FLAG_CAT1 (C++ enumerator), 231
- ESP_AVRC_FEAT_FLAG_CAT2 (C++ enumerator), 231
- ESP_AVRC_FEAT_FLAG_CAT3 (C++ enumerator), 231
- ESP_AVRC_FEAT_FLAG_CAT4 (C++ enumerator), 231
- ESP_AVRC_FEAT_FLAG_COVER_ART_GET_IMAGE (C++ enumerator), 231
- ESP_AVRC_FEAT_FLAG_COVER_ART_GET_IMAGE_PROP (C++ enumerator), 231
- ESP_AVRC_FEAT_FLAG_COVER_ART_GET_LINKED_THUMBNAIL (C++ enumerator), 231
- ESP_AVRC_FEAT_META_DATA (C++ enumerator), 230
- ESP_AVRC_FEAT_RCCT (C++ enumerator), 230
- ESP_AVRC_FEAT_RCTG (C++ enumerator), 230
- ESP_AVRC_FEAT_VENDOR (C++ enumerator), 230
- esp_avrc_feature_flag_t (C++ enum), 231
- esp_avrc_features_t (C++ enum), 230
- ESP_AVRC_MD_ATTR_ALBUM (C++ enumerator), 235
- ESP_AVRC_MD_ATTR_ARTIST (C++ enumerator), 235
- ESP_AVRC_MD_ATTR_GENRE (C++ enumerator), 235
- esp_avrc_md_attr_mask_t (C++ enum), 235
- ESP_AVRC_MD_ATTR_NUM_TRACKS (C++ enumerator), 235
- ESP_AVRC_MD_ATTR_PLAYING_TIME (C++ enumerator), 235
- ESP_AVRC_MD_ATTR_TITLE (C++ enumerator), 235
- ESP_AVRC_MD_ATTR_TRACK_NUM (C++ enumerator), 235
- ESP_AVRC_PLAYBACK_ERROR (C++ enumerator), 238
- ESP_AVRC_PLAYBACK_FWD_SEEK (C++ enumerator), 238
- ESP_AVRC_PLAYBACK_PAUSED (C++ enumerator), 238
- ESP_AVRC_PLAYBACK_PLAYING (C++ enumerator), 238
- ESP_AVRC_PLAYBACK_REV_SEEK (C++ enumerator), 238
- esp_avrc_playback_stat_t (C++ enum), 238
- ESP_AVRC_PLAYBACK_STOPPED (C++ enumerator), 238

- `esp_avrc_ps_attr_ids_t` (C++ *enum*), 236
- `esp_avrc_ps_eq_value_ids_t` (C++ *enum*), 236
- `ESP_AVRC_PS_EQUALIZER` (C++ *enumerator*), 236
- `ESP_AVRC_PS_EQUALIZER_OFF` (C++ *enumerator*), 236
- `ESP_AVRC_PS_EQUALIZER_ON` (C++ *enumerator*), 236
- `ESP_AVRC_PS_MAX_ATTR` (C++ *enumerator*), 236
- `ESP_AVRC_PS_REPEAT_GROUP` (C++ *enumerator*), 237
- `ESP_AVRC_PS_REPEAT_MODE` (C++ *enumerator*), 236
- `ESP_AVRC_PS_REPEAT_OFF` (C++ *enumerator*), 237
- `ESP_AVRC_PS_REPEAT_SINGLE` (C++ *enumerator*), 237
- `esp_avrc_ps_rpt_value_ids_t` (C++ *enum*), 236
- `ESP_AVRC_PS_SCAN_ALL` (C++ *enumerator*), 237
- `ESP_AVRC_PS_SCAN_GROUP` (C++ *enumerator*), 237
- `ESP_AVRC_PS_SCAN_MODE` (C++ *enumerator*), 236
- `ESP_AVRC_PS_SCAN_OFF` (C++ *enumerator*), 237
- `esp_avrc_ps_scn_value_ids_t` (C++ *enum*), 237
- `esp_avrc_ps_shf_value_ids_t` (C++ *enum*), 237
- `ESP_AVRC_PS_SHUFFLE_ALL` (C++ *enumerator*), 237
- `ESP_AVRC_PS_SHUFFLE_GROUP` (C++ *enumerator*), 237
- `ESP_AVRC_PS_SHUFFLE_MODE` (C++ *enumerator*), 236
- `ESP_AVRC_PS_SHUFFLE_OFF` (C++ *enumerator*), 237
- `esp_avrc_psth_bit_mask_operation` (C++ *function*), 224
- `esp_avrc_psth_bit_mask_t` (C++ *class*), 229
- `esp_avrc_psth_bit_mask_t::bits` (C++ *member*), 229
- `ESP_AVRC_PSTH_FILTER_ALLOWED_CMD` (C++ *enumerator*), 233
- `ESP_AVRC_PSTH_FILTER_SUPPORT_MAX` (C++ *enumerator*), 234
- `ESP_AVRC_PSTH_FILTER_SUPPORTED_CMD` (C++ *enumerator*), 234
- `esp_avrc_psth_filter_t` (C++ *enum*), 233
- `ESP_AVRC_PT_CMD_0` (C++ *enumerator*), 232
- `ESP_AVRC_PT_CMD_1` (C++ *enumerator*), 232
- `ESP_AVRC_PT_CMD_2` (C++ *enumerator*), 232
- `ESP_AVRC_PT_CMD_3` (C++ *enumerator*), 232
- `ESP_AVRC_PT_CMD_4` (C++ *enumerator*), 232
- `ESP_AVRC_PT_CMD_5` (C++ *enumerator*), 232
- `ESP_AVRC_PT_CMD_6` (C++ *enumerator*), 232
- `ESP_AVRC_PT_CMD_7` (C++ *enumerator*), 232
- `ESP_AVRC_PT_CMD_8` (C++ *enumerator*), 232
- `ESP_AVRC_PT_CMD_9` (C++ *enumerator*), 232
- `ESP_AVRC_PT_CMD_ANGLE` (C++ *enumerator*), 233
- `ESP_AVRC_PT_CMD_BACKWARD` (C++ *enumerator*), 233
- `ESP_AVRC_PT_CMD_CHAN_DOWN` (C++ *enumerator*), 232
- `ESP_AVRC_PT_CMD_CHAN_UP` (C++ *enumerator*), 232
- `ESP_AVRC_PT_CMD_CLEAR` (C++ *enumerator*), 232
- `ESP_AVRC_PT_CMD_CONT_MENU` (C++ *enumerator*), 231
- `ESP_AVRC_PT_CMD_DISP_INFO` (C++ *enumerator*), 232
- `ESP_AVRC_PT_CMD_DOT` (C++ *enumerator*), 232
- `ESP_AVRC_PT_CMD_DOWN` (C++ *enumerator*), 231
- `ESP_AVRC_PT_CMD_EJECT` (C++ *enumerator*), 233
- `ESP_AVRC_PT_CMD_ENTER` (C++ *enumerator*), 232
- `ESP_AVRC_PT_CMD_EXIT` (C++ *enumerator*), 232
- `ESP_AVRC_PT_CMD_F1` (C++ *enumerator*), 233
- `ESP_AVRC_PT_CMD_F2` (C++ *enumerator*), 233
- `ESP_AVRC_PT_CMD_F3` (C++ *enumerator*), 233
- `ESP_AVRC_PT_CMD_F4` (C++ *enumerator*), 233
- `ESP_AVRC_PT_CMD_F5` (C++ *enumerator*), 233
- `ESP_AVRC_PT_CMD_FAST_FORWARD` (C++ *enumerator*), 233
- `ESP_AVRC_PT_CMD_FAV_MENU` (C++ *enumerator*), 231
- `ESP_AVRC_PT_CMD_FORWARD` (C++ *enumerator*), 233
- `ESP_AVRC_PT_CMD_HELP` (C++ *enumerator*), 232
- `ESP_AVRC_PT_CMD_INPUT_SEL` (C++ *enumerator*), 232
- `ESP_AVRC_PT_CMD_LEFT` (C++ *enumerator*), 231
- `ESP_AVRC_PT_CMD_LEFT_DOWN` (C++ *enumerator*), 231
- `ESP_AVRC_PT_CMD_LEFT_UP` (C++ *enumerator*), 231
- `ESP_AVRC_PT_CMD_MUTE` (C++ *enumerator*), 233
- `ESP_AVRC_PT_CMD_PAGE_DOWN` (C++ *enumerator*), 232
- `ESP_AVRC_PT_CMD_PAGE_UP` (C++ *enumerator*), 232
- `ESP_AVRC_PT_CMD_PAUSE` (C++ *enumerator*), 233
- `ESP_AVRC_PT_CMD_PLAY` (C++ *enumerator*), 233
- `ESP_AVRC_PT_CMD_POWER` (C++ *enumerator*), 233
- `ESP_AVRC_PT_CMD_PREV_CHAN` (C++ *enumerator*), 232
- `ESP_AVRC_PT_CMD_RECORD` (C++ *enumerator*), 233
- `ESP_AVRC_PT_CMD_REWIND` (C++ *enumerator*), 233
- `ESP_AVRC_PT_CMD_RIGHT` (C++ *enumerator*), 231
- `ESP_AVRC_PT_CMD_RIGHT_DOWN` (C++ *enumerator*), 231
- `ESP_AVRC_PT_CMD_RIGHT_UP` (C++ *enumerator*), 231
- `ESP_AVRC_PT_CMD_ROOT_MENU` (C++ *enumerator*), 231
- `ESP_AVRC_PT_CMD_SELECT` (C++ *enumerator*),

- 231
- ESP_AVRC_PT_CMD_SETUP_MENU (C++ *enumerator*), 231
- ESP_AVRC_PT_CMD_SOUND_SEL (C++ *enumerator*), 232
- ESP_AVRC_PT_CMD_STATE_PRESSED (C++ *enumerator*), 234
- ESP_AVRC_PT_CMD_STATE_RELEASED (C++ *enumerator*), 234
- esp_avrc_pt_cmd_state_t (C++ *enum*), 234
- ESP_AVRC_PT_CMD_STOP (C++ *enumerator*), 233
- ESP_AVRC_PT_CMD_SUBPICKT (C++ *enumerator*), 233
- esp_avrc_pt_cmd_t (C++ *enum*), 231
- ESP_AVRC_PT_CMD_UP (C++ *enumerator*), 231
- ESP_AVRC_PT_CMD_VENDOR (C++ *enumerator*), 233
- ESP_AVRC_PT_CMD_VOL_DOWN (C++ *enumerator*), 233
- ESP_AVRC_PT_CMD_VOL_UP (C++ *enumerator*), 233
- ESP_AVRC_RN_ADDRESSED_PLAYER_CHANGE (C++ *enumerator*), 236
- ESP_AVRC_RN_APP_SETTING_CHANGE (C++ *enumerator*), 235
- ESP_AVRC_RN_AVAILABLE_PLAYERS_CHANGE (C++ *enumerator*), 235
- ESP_AVRC_RN_BATTERY_STATUS_CHANGE (C++ *enumerator*), 235
- ESP_AVRC_RN_CAP_ALLOWED_EVT (C++ *enumerator*), 236
- ESP_AVRC_RN_CAP_MAX (C++ *enumerator*), 236
- ESP_AVRC_RN_CAP_SUPPORTED_EVT (C++ *enumerator*), 236
- esp_avrc_rn_event_ids_t (C++ *enum*), 235
- esp_avrc_rn_evt_bit_mask_operation (C++ *function*), 225
- esp_avrc_rn_evt_cap_mask_t (C++ *class*), 229
- esp_avrc_rn_evt_cap_mask_t::bits (C++ *member*), 230
- esp_avrc_rn_evt_cap_t (C++ *enum*), 236
- ESP_AVRC_RN_MAX_EVT (C++ *enumerator*), 236
- ESP_AVRC_RN_NOW_PLAYING_CHANGE (C++ *enumerator*), 235
- esp_avrc_rn_param_t (C++ *union*), 226
- esp_avrc_rn_param_t::batt (C++ *member*), 226
- esp_avrc_rn_param_t::elm_id (C++ *member*), 226
- esp_avrc_rn_param_t::play_pos (C++ *member*), 226
- esp_avrc_rn_param_t::playback (C++ *member*), 226
- esp_avrc_rn_param_t::volume (C++ *member*), 226
- ESP_AVRC_RN_PLAY_POS_CHANGED (C++ *enumerator*), 235
- ESP_AVRC_RN_PLAY_STATUS_CHANGE (C++ *enumerator*), 235
- ESP_AVRC_RN_RSP_CHANGED (C++ *enumerator*), 236
- ESP_AVRC_RN_RSP_INTERIM (C++ *enumerator*), 236
- esp_avrc_rn_rsp_t (C++ *enum*), 236
- ESP_AVRC_RN_SYSTEM_STATUS_CHANGE (C++ *enumerator*), 235
- ESP_AVRC_RN_TRACK_CHANGE (C++ *enumerator*), 235
- ESP_AVRC_RN_TRACK_REACHED_END (C++ *enumerator*), 235
- ESP_AVRC_RN_TRACK_REACHED_START (C++ *enumerator*), 235
- ESP_AVRC_RN_UIDS_CHANGE (C++ *enumerator*), 236
- ESP_AVRC_RN_VOLUME_CHANGE (C++ *enumerator*), 236
- ESP_AVRC_RSP_ACCEPT (C++ *enumerator*), 237
- ESP_AVRC_RSP_CHANGED (C++ *enumerator*), 237
- ESP_AVRC_RSP_IMPL_STBL (C++ *enumerator*), 237
- ESP_AVRC_RSP_IN_TRANS (C++ *enumerator*), 237
- ESP_AVRC_RSP_INTERIM (C++ *enumerator*), 237
- ESP_AVRC_RSP_NOT_IMPL (C++ *enumerator*), 237
- ESP_AVRC_RSP_REJECT (C++ *enumerator*), 237
- esp_avrc_rsp_t (C++ *enum*), 237
- esp_avrc_set_app_value_param_t (C++ *class*), 230
- esp_avrc_set_app_value_param_t::attr_id (C++ *member*), 230
- esp_avrc_set_app_value_param_t::attr_val (C++ *member*), 230
- esp_avrc_tg_cb_event_t (C++ *enum*), 234
- esp_avrc_tg_cb_param_t (C++ *union*), 228
- esp_avrc_tg_cb_param_t::avrc_tg_conn_stat_param (C++ *class*), 228
- esp_avrc_tg_cb_param_t::avrc_tg_conn_stat_param:: (C++ *member*), 228
- esp_avrc_tg_cb_param_t::avrc_tg_conn_stat_param:: (C++ *member*), 228
- esp_avrc_tg_cb_param_t::avrc_tg_psth_cmd_param (C++ *class*), 228
- esp_avrc_tg_cb_param_t::avrc_tg_psth_cmd_param::k (C++ *member*), 228
- esp_avrc_tg_cb_param_t::avrc_tg_psth_cmd_param::k (C++ *member*), 228
- esp_avrc_tg_cb_param_t::avrc_tg_reg_ntf_param (C++ *class*), 228
- esp_avrc_tg_cb_param_t::avrc_tg_reg_ntf_param::ev (C++ *member*), 229
- esp_avrc_tg_cb_param_t::avrc_tg_reg_ntf_param::ev (C++ *member*), 229
- esp_avrc_tg_cb_param_t::avrc_tg_rmt_feats_param (C++ *class*), 229
- esp_avrc_tg_cb_param_t::avrc_tg_rmt_feats_param:: (C++ *member*), 229

- ESP_BLE_AD_TYPE_SM_TK (C++ enumerator), 148
- ESP_BLE_AD_TYPE_SOL_SRV_UUID (C++ enumerator), 148
- ESP_BLE_AD_TYPE_SPAIR_C256 (C++ enumerator), 148
- ESP_BLE_AD_TYPE_SPAIR_R256 (C++ enumerator), 148
- ESP_BLE_AD_TYPE_TRANS_DISC_DATA (C++ enumerator), 148
- ESP_BLE_AD_TYPE_TX_PWR (C++ enumerator), 147
- ESP_BLE_AD_TYPE_URI (C++ enumerator), 148
- esp_ble_addr_type_t (C++ enum), 123
- esp_ble_adv_channel_t (C++ enum), 148
- ESP_BLE_ADV_DATA_LEN_MAX (C macro), 145
- esp_ble_adv_data_t (C++ class), 138
- esp_ble_adv_data_t::appearance (C++ member), 138
- esp_ble_adv_data_t::flag (C++ member), 138
- esp_ble_adv_data_t::include_name (C++ member), 138
- esp_ble_adv_data_t::include_txpower (C++ member), 138
- esp_ble_adv_data_t::manufacturer_len (C++ member), 138
- esp_ble_adv_data_t::max_interval (C++ member), 138
- esp_ble_adv_data_t::min_interval (C++ member), 138
- esp_ble_adv_data_t::p_manufacturer_data (C++ member), 138
- esp_ble_adv_data_t::p_service_data (C++ member), 138
- esp_ble_adv_data_t::p_service_uuid (C++ member), 138
- esp_ble_adv_data_t::service_data_len (C++ member), 138
- esp_ble_adv_data_t::service_uuid_len (C++ member), 138
- esp_ble_adv_data_t::set_scan_rsp (C++ member), 138
- esp_ble_adv_data_type (C++ enum), 147
- esp_ble_adv_filter_t (C++ enum), 149
- ESP_BLE_ADV_FLAG_BREDR_NOT_SPT (C macro), 143
- ESP_BLE_ADV_FLAG_DMT_CONTROLLER_SPT (C macro), 143
- ESP_BLE_ADV_FLAG_DMT_HOST_SPT (C macro), 143
- ESP_BLE_ADV_FLAG_GEN_DISC (C macro), 143
- ESP_BLE_ADV_FLAG_LIMIT_DISC (C macro), 143
- ESP_BLE_ADV_FLAG_NON_LIMIT_DISC (C macro), 143
- esp_ble_adv_params_t (C++ class), 137
- esp_ble_adv_params_t::adv_filter_policy (C++ member), 138
- esp_ble_adv_params_t::adv_int_max (C++ member), 137
- esp_ble_adv_params_t::adv_int_min (C++ member), 137
- esp_ble_adv_params_t::adv_type (C++ member), 137
- esp_ble_adv_params_t::channel_map (C++ member), 138
- esp_ble_adv_params_t::own_addr_type (C++ member), 138
- esp_ble_adv_params_t::peer_addr (C++ member), 138
- esp_ble_adv_params_t::peer_addr_type (C++ member), 138
- esp_ble_adv_type_t (C++ enum), 148
- ESP_BLE_APP_ENC_KEY_SIZE (C++ enumerator), 149
- ESP_BLE_APPEARANCE_BLOOD_PRESSURE_ARM (C macro), 144
- ESP_BLE_APPEARANCE_BLOOD_PRESSURE_WRIST (C macro), 144
- ESP_BLE_APPEARANCE_CYCLING_CADENCE (C macro), 145
- ESP_BLE_APPEARANCE_CYCLING_COMPUTER (C macro), 145
- ESP_BLE_APPEARANCE_CYCLING_POWER (C macro), 145
- ESP_BLE_APPEARANCE_CYCLING_SPEED (C macro), 145
- ESP_BLE_APPEARANCE_CYCLING_SPEED_CADENCE (C macro), 145
- ESP_BLE_APPEARANCE_GENERIC_BARCODE_SCANNER (C macro), 144
- ESP_BLE_APPEARANCE_GENERIC_BLOOD_PRESSURE (C macro), 144
- ESP_BLE_APPEARANCE_GENERIC_CLOCK (C macro), 144
- ESP_BLE_APPEARANCE_GENERIC_COMPUTER (C macro), 144
- ESP_BLE_APPEARANCE_GENERIC_CONTINUOUS_GLUCOSE_MONITORING (C macro), 145
- ESP_BLE_APPEARANCE_GENERIC_CYCLING (C macro), 145
- ESP_BLE_APPEARANCE_GENERIC_DISPLAY (C macro), 144
- ESP_BLE_APPEARANCE_GENERIC_EYEGLASSES (C macro), 144
- ESP_BLE_APPEARANCE_GENERIC_GLUCOSE (C macro), 144
- ESP_BLE_APPEARANCE_GENERIC_HEART_RATE (C macro), 144
- ESP_BLE_APPEARANCE_GENERIC_HID (C macro), 144
- ESP_BLE_APPEARANCE_GENERIC_INSULIN_PUMP (C macro), 145
- ESP_BLE_APPEARANCE_GENERIC_KEYRING (C macro), 144
- ESP_BLE_APPEARANCE_GENERIC_MEDIA_PLAYER

- (*C macro*), 144
- ESP_BLE_APPEARANCE_GENERIC_MEDICATION_DESCRIPTOR (*C macro*), 145
- ESP_BLE_APPEARANCE_GENERIC_OUTDOOR_SPORTS_LOCATION (*C macro*), 145
- ESP_BLE_APPEARANCE_GENERIC_PERSONAL_MOBILITY_DEVICE (*C macro*), 145
- ESP_BLE_APPEARANCE_GENERIC_PHONE (*C macro*), 144
- ESP_BLE_APPEARANCE_GENERIC_PULSE_OXIMETER (*C macro*), 145
- ESP_BLE_APPEARANCE_GENERIC_REMOTE (*C macro*), 144
- ESP_BLE_APPEARANCE_GENERIC_TAG (*C macro*), 144
- ESP_BLE_APPEARANCE_GENERIC_THERMOMETER (*C macro*), 144
- ESP_BLE_APPEARANCE_GENERIC_WALKING (*C macro*), 144
- ESP_BLE_APPEARANCE_GENERIC_WATCH (*C macro*), 144
- ESP_BLE_APPEARANCE_GENERIC_WEIGHT (*C macro*), 145
- ESP_BLE_APPEARANCE_HEART_RATE_BELT (*C macro*), 144
- ESP_BLE_APPEARANCE_HID_BARCODE_SCANNER (*C macro*), 144
- ESP_BLE_APPEARANCE_HID_CARD_READER (*C macro*), 144
- ESP_BLE_APPEARANCE_HID_DIGITAL_PEN (*C macro*), 144
- ESP_BLE_APPEARANCE_HID_DIGITIZER_TABLET (*C macro*), 144
- ESP_BLE_APPEARANCE_HID_GAMEPAD (*C macro*), 144
- ESP_BLE_APPEARANCE_HID_JOYSTICK (*C macro*), 144
- ESP_BLE_APPEARANCE_HID_KEYBOARD (*C macro*), 144
- ESP_BLE_APPEARANCE_HID_MOUSE (*C macro*), 144
- ESP_BLE_APPEARANCE_INSULIN_PEN (*C macro*), 145
- ESP_BLE_APPEARANCE_INSULIN_PUMP_DURABLE_PUMP (*C macro*), 145
- ESP_BLE_APPEARANCE_INSULIN_PUMP_PATCH_PUMP (*C macro*), 145
- ESP_BLE_APPEARANCE_MOBILITY_SCOOTER (*C macro*), 145
- ESP_BLE_APPEARANCE_OUTDOOR_SPORTS_LOCATION (*C macro*), 145
- ESP_BLE_APPEARANCE_OUTDOOR_SPORTS_LOCATION (*C macro*), 145
- ESP_BLE_APPEARANCE_OUTDOOR_SPORTS_LOCATION (*C macro*), 145
- ESP_BLE_APPEARANCE_OUTDOOR_SPORTS_LOCATION (*C macro*), 145
- ESP_BLE_APPEARANCE_POWERED_WHEELCHAIR (*C macro*), 145
- (*C macro*), 145
- ESP_BLE_APPEARANCE_PULSE_OXIMETER_FINGERTIP (*C macro*), 145
- ESP_BLE_APPEARANCE_PULSE_OXIMETER_WRIST (*C macro*), 145
- ESP_BLE_APPEARANCE_SPORTS_WATCH (*C macro*), 144
- ESP_BLE_APPEARANCE_THERMOMETER_EAR (*C macro*), 144
- ESP_BLE_APPEARANCE_UNKNOWN (*C macro*), 144
- ESP_BLE_APPEARANCE_WALKING_IN_SHOE (*C macro*), 145
- ESP_BLE_APPEARANCE_WALKING_ON_HIP (*C macro*), 145
- ESP_BLE_APPEARANCE_WALKING_ON_SHOE (*C macro*), 145
- esp_ble_auth_cmpl_t (*C++ class*), 142
- esp_ble_auth_cmpl_t::addr_type (*C++ member*), 143
- esp_ble_auth_cmpl_t::auth_mode (*C++ member*), 143
- esp_ble_auth_cmpl_t::bd_addr (*C++ member*), 142
- esp_ble_auth_cmpl_t::dev_type (*C++ member*), 143
- esp_ble_auth_cmpl_t::fail_reason (*C++ member*), 143
- esp_ble_auth_cmpl_t::key (*C++ member*), 143
- esp_ble_auth_cmpl_t::key_present (*C++ member*), 142
- esp_ble_auth_cmpl_t::key_type (*C++ member*), 143
- esp_ble_auth_cmpl_t::success (*C++ member*), 143
- esp_ble_auth_req_t (*C++ type*), 146
- esp_ble_bond_dev_t (*C++ class*), 142
- esp_ble_bond_dev_t::bd_addr (*C++ member*), 142
- esp_ble_bond_dev_t::bond_key (*C++ member*), 142
- esp_ble_bond_key_info_t (*C++ class*), 141
- esp_ble_bond_key_info_t::key_mask (*C++ member*), 142
- esp_ble_bond_key_info_t::pcsrk_key (*C++ member*), 142
- esp_ble_bond_key_info_t::penc_key (*C++ member*), 142
- esp_ble_bond_key_info_t::pid_key (*C++ member*), 142
- esp_ble_confirm_reply (*C++ function*), 130
- ESP_BLE_CONN_PARAM_DEF (*C macro*), 121
- ESP_BLE_CONN_INT_MIN (*C macro*), 121
- ESP_BLE_CONN_LATENCY_MAX (*C macro*), 121
- ESP_BLE_CONN_PARAM_UNDEF (*C macro*), 121
- ESP_BLE_CONN_SUP_TOUT_MAX (*C macro*), 121
- ESP_BLE_CONN_SUP_TOUT_MIN (*C macro*), 121
- esp_ble_conn_update_params_t (*C++ class*),

- (C++ member), 188
- esp_ble_gattc_cb_param_t::gattc_write_evt_param (C++ class), 188
- esp_ble_gattc_cb_param_t::gattc_write_evt_param (C++ member), 188
- esp_ble_gattc_cb_param_t::gattc_write_evt_param::handle (C++ member), 188
- esp_ble_gattc_cb_param_t::gattc_write_evt_param::set_ad_char_descr (C++ member), 188
- esp_ble_gattc_cb_param_t::gattc_write_evt_param::status_read_multiple (C++ member), 188
- esp_ble_gattc_cb_param_t::get_addr_list (C++ member), 183
- esp_ble_gattc_cb_param_t::notify (C++ member), 183
- esp_ble_gattc_cb_param_t::open (C++ member), 183
- esp_ble_gattc_cb_param_t::queue_full (C++ member), 183
- esp_ble_gattc_cb_param_t::read (C++ member), 183
- esp_ble_gattc_cb_param_t::reg (C++ member), 183
- esp_ble_gattc_cb_param_t::reg_for_notify (C++ member), 183
- esp_ble_gattc_cb_param_t::search_cmpl (C++ member), 183
- esp_ble_gattc_cb_param_t::search_res (C++ member), 183
- esp_ble_gattc_cb_param_t::set_assoc_cmp (C++ member), 183
- esp_ble_gattc_cb_param_t::srvc_chg (C++ member), 183
- esp_ble_gattc_cb_param_t::unreg_for_notify (C++ member), 183
- esp_ble_gattc_cb_param_t::write (C++ member), 183
- esp_ble_gattc_close (C++ function), 176
- esp_ble_gattc_execute_write (C++ function), 181
- esp_ble_gattc_get_all_char (C++ function), 176
- esp_ble_gattc_get_all_descr (C++ function), 177
- esp_ble_gattc_get_attr_count (C++ function), 179
- esp_ble_gattc_get_char_by_uuid (C++ function), 177
- esp_ble_gattc_get_db (C++ function), 179
- esp_ble_gattc_get_descr_by_char_handle (C++ function), 178
- esp_ble_gattc_get_descr_by_uuid (C++ function), 177
- esp_ble_gattc_get_include_service (C++ function), 178
- esp_ble_gattc_get_service (C++ function), 176
- esp_ble_gattc_open (C++ function), 175
- esp_ble_gattc_prepare_write (C++ function), 181
- esp_ble_gattc_prepare_write_char_descr (C++ function), 181
- esp_ble_gattc_read_by_type (C++ function), 179
- esp_ble_gattc_read_char (C++ function), 179
- esp_ble_gattc_read_char_descr (C++ function), 180
- esp_ble_gattc_read_multiple (C++ function), 180
- esp_ble_gattc_register_callback (C++ function), 175
- esp_ble_gattc_register_for_notify (C++ function), 181
- esp_ble_gattc_search_service (C++ function), 176
- esp_ble_gattc_send_mtu_req (C++ function), 176
- esp_ble_gattc_unregister_for_notify (C++ function), 182
- esp_ble_gattc_write_char (C++ function), 180
- esp_ble_gattc_write_char_descr (C++ function), 180
- esp_ble_gatts_add_char (C++ function), 164
- esp_ble_gatts_add_char_descr (C++ function), 164
- esp_ble_gatts_add_included_service (C++ function), 164
- esp_ble_gatts_app_register (C++ function), 163
- esp_ble_gatts_app_unregister (C++ function), 163
- esp_ble_gatts_cb_param_t (C++ union), 166
- esp_ble_gatts_cb_param_t::add_attr_tab (C++ member), 167
- esp_ble_gatts_cb_param_t::add_char (C++ member), 167
- esp_ble_gatts_cb_param_t::add_char_descr (C++ member), 167
- esp_ble_gatts_cb_param_t::add_incl_srvc (C++ member), 166
- esp_ble_gatts_cb_param_t::cancel_open (C++ member), 167
- esp_ble_gatts_cb_param_t::close (C++ member), 167
- esp_ble_gatts_cb_param_t::conf (C++ member), 166
- esp_ble_gatts_cb_param_t::congest (C++ member), 167
- esp_ble_gatts_cb_param_t::connect (C++ member), 167
- esp_ble_gatts_cb_param_t::create (C++ member), 166
- esp_ble_gatts_cb_param_t::del (C++ member), 167
- esp_ble_gatts_cb_param_t::disconnect

- tion*), 165
- esp_ble_gatts_start_service (C++ *function*), 164
- esp_ble_gatts_stop_service (C++ *function*), 165
- esp_ble_get_bond_device_list (C++ *function*), 130
- esp_ble_get_bond_device_num (C++ *function*), 130
- esp_ble_get_current_conn_params (C++ *function*), 131
- ESP_BLE_HOST_STATUS_CHECK (C *macro*), 308
- ESP_BLE_HOST_STATUS_ENABLED (C *macro*), 308
- ESP_BLE_ID_KEY_MASK (C *macro*), 121
- esp_ble_io_cap_t (C++ *type*), 146
- ESP_BLE_IS_VALID_PARAM (C *macro*), 121
- esp_ble_key_mask_t (C++ *type*), 122
- esp_ble_key_t (C++ *class*), 142
- esp_ble_key_t::bd_addr (C++ *member*), 142
- esp_ble_key_t::key_type (C++ *member*), 142
- esp_ble_key_t::p_key_value (C++ *member*), 142
- esp_ble_key_type_t (C++ *type*), 146
- esp_ble_key_value_t (C++ *union*), 131
- esp_ble_key_value_t::lcsr_key (C++ *member*), 131
- esp_ble_key_value_t::lenc_key (C++ *member*), 131
- esp_ble_key_value_t::pcsr_key (C++ *member*), 131
- esp_ble_key_value_t::penc_key (C++ *member*), 131
- esp_ble_key_value_t::pid_key (C++ *member*), 131
- esp_ble_lcsr_keys (C++ *class*), 141
- esp_ble_lcsr_keys::counter (C++ *member*), 141
- esp_ble_lcsr_keys::csr_key (C++ *member*), 141
- esp_ble_lcsr_keys::div (C++ *member*), 141
- esp_ble_lcsr_keys::sec_level (C++ *member*), 141
- esp_ble_lenc_keys_t (C++ *class*), 141
- esp_ble_lenc_keys_t::div (C++ *member*), 141
- esp_ble_lenc_keys_t::key_size (C++ *member*), 141
- esp_ble_lenc_keys_t::ltk (C++ *member*), 141
- esp_ble_lenc_keys_t::sec_level (C++ *member*), 141
- ESP_BLE_LINK_KEY_MASK (C *macro*), 121
- esp_ble_local_id_keys_t (C++ *class*), 142
- esp_ble_local_id_keys_t::dhk (C++ *member*), 142
- esp_ble_local_id_keys_t::ir (C++ *member*), 142
- esp_ble_local_id_keys_t::irk (C++ *member*), 142
- ESP_BLE_MESH_ACTUATOR_BLOCKED_ERROR (C *macro*), 409
- ESP_BLE_MESH_ACTUATOR_BLOCKED_WARNING (C *macro*), 409
- ESP_BLE_MESH_ADDR_ALL_NODES (C *macro*), 309
- ESP_BLE_MESH_ADDR_FRIENDS (C *macro*), 309
- ESP_BLE_MESH_ADDR_IS_GROUP (C *macro*), 309
- ESP_BLE_MESH_ADDR_IS_RFU (C *macro*), 309
- ESP_BLE_MESH_ADDR_IS_UNICAST (C *macro*), 309
- ESP_BLE_MESH_ADDR_IS_VIRTUAL (C *macro*), 309
- ESP_BLE_MESH_ADDR_PROXIES (C *macro*), 309
- ESP_BLE_MESH_ADDR_RELAYS (C *macro*), 309
- ESP_BLE_MESH_ADDR_TYPE_PUBLIC (C *macro*), 311
- ESP_BLE_MESH_ADDR_TYPE_RANDOM (C *macro*), 311
- ESP_BLE_MESH_ADDR_TYPE_RPA_PUBLIC (C *macro*), 311
- ESP_BLE_MESH_ADDR_TYPE_RPA_RANDOM (C *macro*), 311
- esp_ble_mesh_addr_type_t (C++ *type*), 323
- ESP_BLE_MESH_ADDR_UNASSIGNED (C *macro*), 308
- ESP_BLE_MESH_BATTERY_LOW_ERROR (C *macro*), 408
- ESP_BLE_MESH_BATTERY_LOW_WARNING (C *macro*), 408
- esp_ble_mesh_bd_addr_t (C++ *type*), 323
- ESP_BLE_MESH_BEACON_DISABLED (C *macro*), 309
- ESP_BLE_MESH_BEACON_ENABLED (C *macro*), 309
- ESP_BLE_MESH_BEEP (C++ *enumerator*), 324
- ESP_BLE_MESH_BLINK (C++ *enumerator*), 324
- esp_ble_mesh_cb_t (C++ *type*), 323
- esp_ble_mesh_cb_type_t (C++ *enum*), 324
- esp_ble_mesh_cfg_app_key_add_t (C++ *class*), 356
- esp_ble_mesh_cfg_app_key_add_t::app_idx (C++ *member*), 356
- esp_ble_mesh_cfg_app_key_add_t::app_key (C++ *member*), 356
- esp_ble_mesh_cfg_app_key_add_t::net_idx (C++ *member*), 356
- esp_ble_mesh_cfg_app_key_delete_t (C++ *class*), 360
- esp_ble_mesh_cfg_app_key_delete_t::app_idx (C++ *member*), 360
- esp_ble_mesh_cfg_app_key_delete_t::net_idx (C++ *member*), 360
- esp_ble_mesh_cfg_app_key_get_t (C++ *class*), 354
- esp_ble_mesh_cfg_app_key_get_t::net_idx

- (C++ member), 354
- esp_ble_mesh_cfg_app_key_list_cb_t (C++ class), 365
- esp_ble_mesh_cfg_app_key_list_cb_t::app_key_list (C++ member), 365
- esp_ble_mesh_cfg_app_key_list_cb_t::net_key_list (C++ member), 365
- esp_ble_mesh_cfg_app_key_list_cb_t::status (C++ member), 365
- esp_ble_mesh_cfg_app_key_status_cb_t (C++ class), 363
- esp_ble_mesh_cfg_app_key_status_cb_t::app_key_list (C++ member), 363
- esp_ble_mesh_cfg_app_key_status_cb_t::net_key_list (C++ member), 363
- esp_ble_mesh_cfg_app_key_status_cb_t::status (C++ member), 363
- esp_ble_mesh_cfg_app_key_update_t (C++ class), 359
- esp_ble_mesh_cfg_app_key_update_t::app_key_list (C++ member), 359
- esp_ble_mesh_cfg_app_key_update_t::app_key_status (C++ member), 359
- esp_ble_mesh_cfg_app_key_update_t::net_key_list (C++ member), 359
- esp_ble_mesh_cfg_beacon_set_t (C++ class), 355
- esp_ble_mesh_cfg_beacon_set_t::beacon (C++ member), 355
- esp_ble_mesh_cfg_beacon_status_cb_t (C++ class), 361
- esp_ble_mesh_cfg_beacon_status_cb_t::beacon (C++ member), 361
- esp_ble_mesh_cfg_client_cb_event_t (C++ enum), 371
- esp_ble_mesh_cfg_client_cb_param_t (C++ class), 366
- esp_ble_mesh_cfg_client_cb_param_t::error_code (C++ member), 367
- esp_ble_mesh_cfg_client_cb_param_t::params (C++ member), 367
- esp_ble_mesh_cfg_client_cb_param_t::status (C++ member), 367
- esp_ble_mesh_cfg_client_cb_t (C++ type), 370
- esp_ble_mesh_cfg_client_common_cb_param_t (C++ union), 350
- esp_ble_mesh_cfg_client_common_cb_param_t::app_key_list (C++ member), 351
- esp_ble_mesh_cfg_client_common_cb_param_t::app_key_status (C++ member), 350
- esp_ble_mesh_cfg_client_common_cb_param_t::beacon_status (C++ member), 350
- esp_ble_mesh_cfg_client_common_cb_param_t::complete_status (C++ member), 350
- esp_ble_mesh_cfg_client_common_cb_param_t::default_status (C++ member), 350
- esp_ble_mesh_cfg_client_common_cb_param_t::ESP_BLE_MESH_CFG_CLIENT_PUBLISH_EVT (C++ member), 350
- esp_ble_mesh_cfg_client_common_cb_param_t::gatt_profile (C++ member), 350
- esp_ble_mesh_cfg_client_common_cb_param_t::heartbeat (C++ member), 351
- esp_ble_mesh_cfg_client_common_cb_param_t::heartbeat_notify (C++ member), 351
- esp_ble_mesh_cfg_client_common_cb_param_t::kr_phase_notify (C++ member), 351
- esp_ble_mesh_cfg_client_common_cb_param_t::lpn_notify (C++ member), 351
- esp_ble_mesh_cfg_client_common_cb_param_t::model_pub_get (C++ member), 351
- esp_ble_mesh_cfg_client_common_cb_param_t::node_identity_notify (C++ member), 351
- esp_ble_mesh_cfg_client_common_cb_param_t::sig_model_appl (C++ member), 351
- esp_ble_mesh_cfg_client_common_cb_param_t::sig_model_su (C++ member), 351
- esp_ble_mesh_cfg_client_common_cb_param_t::vnd_model_appl (C++ member), 351
- esp_ble_mesh_cfg_client_common_cb_param_t::vnd_model_su (C++ member), 351
- ESP_BLE_MESH_CFG_CLIENT_GET_STATE_EVT (C++ enumerator), 371
- ESP_BLE_MESH_CFG_CLIENT_EVT_MAX (C++ enumerator), 371
- esp_ble_mesh_cfg_client_get_state_t (C++ union), 348
- esp_ble_mesh_cfg_client_get_state_t::app_key_get (C++ member), 348
- esp_ble_mesh_cfg_client_get_state_t::comp_data_get (C++ member), 348
- esp_ble_mesh_cfg_client_get_state_t::kr_phase_get (C++ member), 348
- esp_ble_mesh_cfg_client_get_state_t::lpn_notify (C++ member), 348
- esp_ble_mesh_cfg_client_get_state_t::model_pub_get (C++ member), 348
- esp_ble_mesh_cfg_client_get_state_t::node_identity_notify (C++ member), 348
- esp_ble_mesh_cfg_client_get_state_t::sig_model_appl (C++ member), 348
- esp_ble_mesh_cfg_client_get_state_t::sig_model_su (C++ member), 348
- esp_ble_mesh_cfg_client_get_state_t::vnd_model_appl (C++ member), 348
- esp_ble_mesh_cfg_client_get_state_t::vnd_model_su (C++ member), 348
- ESP_BLE_MESH_CFG_CLIENT_PUBLISH_EVT (C++ enumerator), 371

- (C++ enumerator), 371
- ESP_BLE_MESH_CFG_CLIENT_SET_STATE_EVT (C++ enumerator), 371
- esp_ble_mesh_cfg_client_set_state_t (C++ union), 349
- esp_ble_mesh_cfg_client_set_state_t::app_key_id (C++ member), 349
- esp_ble_mesh_cfg_client_set_state_t::app_key_id_list (C++ member), 350
- esp_ble_mesh_cfg_client_set_state_t::app_key_update (C++ member), 350
- esp_ble_mesh_cfg_client_set_state_t::beacon (C++ member), 349
- esp_ble_mesh_cfg_client_set_state_t::default_ttl (C++ member), 349
- esp_ble_mesh_cfg_client_set_state_t::frs (C++ member), 349
- esp_ble_mesh_cfg_client_set_state_t::gatt_proxy (C++ member), 349
- esp_ble_mesh_cfg_client_set_state_t::hb (C++ member), 349
- esp_ble_mesh_cfg_client_set_state_t::hb_friend_set_t (C++ member), 349
- esp_ble_mesh_cfg_client_set_state_t::kpr (C++ member), 350
- esp_ble_mesh_cfg_client_set_state_t::mod (C++ member), 349
- esp_ble_mesh_cfg_client_set_state_t::mod_app (C++ member), 350
- esp_ble_mesh_cfg_client_set_state_t::mod_app_mesh (C++ member), 350
- esp_ble_mesh_cfg_client_set_state_t::mod_app_mesh_cfg (C++ member), 350
- esp_ble_mesh_cfg_client_set_state_t::mod_app_mesh_cfg_gatt_proxy (C++ member), 350
- esp_ble_mesh_cfg_client_set_state_t::mod_app_mesh_cfg_gatt_proxy_status (C++ member), 349
- esp_ble_mesh_cfg_client_set_state_t::mod_app_mesh_cfg_gatt_proxy_status_cb (C++ member), 349
- esp_ble_mesh_cfg_client_set_state_t::mod_app_mesh_cfg_hb (C++ member), 349
- esp_ble_mesh_cfg_client_set_state_t::mod_app_mesh_cfg_hb_status (C++ member), 349
- esp_ble_mesh_cfg_client_set_state_t::mod_app_mesh_cfg_hb_status_cb (C++ member), 349
- esp_ble_mesh_cfg_client_set_state_t::mod_app_mesh_cfg_hb_status_cb::count (C++ member), 349
- esp_ble_mesh_cfg_client_set_state_t::mod_app_mesh_cfg_hb_status_cb::dst (C++ member), 349
- esp_ble_mesh_cfg_client_set_state_t::mod_app_mesh_cfg_hb_status_cb::features (C++ member), 349
- esp_ble_mesh_cfg_client_set_state_t::mod_app_mesh_cfg_hb_status_cb::net_idx (C++ member), 349
- esp_ble_mesh_cfg_client_set_state_t::net_key_id (C++ member), 349
- esp_ble_mesh_cfg_client_set_state_t::net_key_id_list (C++ member), 350
- esp_ble_mesh_cfg_client_set_state_t::net_key_update (C++ member), 350
- esp_ble_mesh_cfg_client_set_state_t::net_tx_params (C++ member), 350
- esp_ble_mesh_cfg_client_set_state_t::net_tx_params_set (C++ member), 350
- esp_ble_mesh_cfg_client_set_state_t::net_tx_params_set::count (C++ member), 350
- esp_ble_mesh_cfg_client_set_state_t::relay (C++ member), 350
- (C++ member), 349
- ESP_BLE_MESH_CFG_CLIENT_TIMEOUT_EVT (C++ enumerator), 371
- esp_ble_mesh_cfg_comp_data_status_cb_t (C++ class), 361
- esp_ble_mesh_cfg_comp_data_status_cb_t::composition_data (C++ member), 361
- esp_ble_mesh_cfg_comp_data_status_cb_t::page (C++ member), 361
- esp_ble_mesh_cfg_composition_data_get_t (C++ class), 353
- esp_ble_mesh_cfg_composition_data_get_t::page (C++ member), 353
- esp_ble_mesh_cfg_default_ttl_set_t (C++ class), 355
- esp_ble_mesh_cfg_default_ttl_set_t::ttl (C++ member), 355
- esp_ble_mesh_cfg_default_ttl_status_cb_t (C++ class), 361
- esp_ble_mesh_cfg_default_ttl_status_cb_t::default (C++ member), 362
- esp_ble_mesh_cfg_friend_set_t (C++ class), 355
- esp_ble_mesh_cfg_friend_set_t::friend_state (C++ member), 355
- esp_ble_mesh_cfg_friend_status_cb_t (C++ class), 364
- esp_ble_mesh_cfg_friend_status_cb_t::friend_state (C++ member), 364
- esp_ble_mesh_cfg_gatt_proxy_set_t (C++ class), 355
- esp_ble_mesh_cfg_gatt_proxy_set_t::gatt_proxy (C++ member), 355
- esp_ble_mesh_cfg_gatt_proxy_status_cb_t (C++ class), 362
- esp_ble_mesh_cfg_gatt_proxy_status_cb_t::gatt_pro (C++ member), 362
- esp_ble_mesh_cfg_hb_pub_status_cb_t (C++ class), 364
- esp_ble_mesh_cfg_hb_pub_status_cb_t::count (C++ member), 364
- esp_ble_mesh_cfg_hb_pub_status_cb_t::dst (C++ member), 364
- esp_ble_mesh_cfg_hb_pub_status_cb_t::features (C++ member), 364
- esp_ble_mesh_cfg_hb_pub_status_cb_t::net_idx (C++ member), 364
- esp_ble_mesh_cfg_hb_pub_status_cb_t::period (C++ member), 364
- esp_ble_mesh_cfg_hb_pub_status_cb_t::status (C++ member), 364
- esp_ble_mesh_cfg_hb_pub_status_cb_t::ttl (C++ member), 364
- esp_ble_mesh_cfg_hb_sub_status_cb_t (C++ class), 364
- esp_ble_mesh_cfg_hb_sub_status_cb_t::count (C++ member), 364
- esp_ble_mesh_cfg_hb_sub_status_cb_t::dst (C++ member), 364

- (C++ member), 357
- esp_ble_mesh_cfg_model_pub_set_t::cred_flag (C++ member), 356
- esp_ble_mesh_cfg_model_pub_set_t::element_addr (C++ member), 356
- esp_ble_mesh_cfg_model_pub_set_t::model_id (C++ member), 357
- esp_ble_mesh_cfg_model_pub_set_t::public_key (C++ member), 356
- esp_ble_mesh_cfg_model_pub_set_t::public_key_index (C++ member), 356
- esp_ble_mesh_cfg_model_pub_set_t::public_key_on (C++ member), 356
- esp_ble_mesh_cfg_model_pub_set_t::public_key_remove (C++ member), 356
- esp_ble_mesh_cfg_model_pub_set_t::public_key_title (C++ member), 356
- esp_ble_mesh_cfg_model_pub_status_cb_t (C++ class), 362
- esp_ble_mesh_cfg_model_pub_status_cb_t::element_addr (C++ member), 362
- esp_ble_mesh_cfg_model_pub_status_cb_t::company_id (C++ member), 362
- esp_ble_mesh_cfg_model_pub_status_cb_t::element_id (C++ member), 362
- esp_ble_mesh_cfg_model_pub_status_cb_t::model_id (C++ member), 362
- esp_ble_mesh_cfg_model_pub_status_cb_t::sub_addr (C++ member), 362
- esp_ble_mesh_cfg_model_pub_status_cb_t::status (C++ member), 362
- esp_ble_mesh_cfg_model_pub_status_cb_t::sub_addr (C++ member), 362
- esp_ble_mesh_cfg_model_pub_status_cb_t::company_id (C++ member), 362
- esp_ble_mesh_cfg_model_pub_status_cb_t::element_addr (C++ member), 362
- esp_ble_mesh_cfg_model_pub_status_cb_t::model_id (C++ member), 362
- esp_ble_mesh_cfg_model_pub_status_cb_t::status (C++ member), 362
- esp_ble_mesh_cfg_model_pub_status_cb_t::sub_addr (C++ member), 362
- esp_ble_mesh_cfg_model_pub_va_set_t (C++ class), 358
- esp_ble_mesh_cfg_model_pub_va_set_t::company_id (C++ member), 359
- esp_ble_mesh_cfg_model_pub_va_set_t::cred_flag (C++ member), 358
- esp_ble_mesh_cfg_model_pub_va_set_t::element_addr (C++ member), 358
- esp_ble_mesh_cfg_model_pub_va_set_t::label (C++ member), 358
- esp_ble_mesh_cfg_model_pub_va_set_t::model_id (C++ member), 359
- esp_ble_mesh_cfg_model_pub_va_set_t::public_key (C++ member), 358
- esp_ble_mesh_cfg_model_pub_va_set_t::public_key_index (C++ member), 359
- esp_ble_mesh_cfg_model_pub_va_set_t::public_key_remove (C++ member), 359
- esp_ble_mesh_cfg_model_pub_va_set_t::public_key_title (C++ member), 359
- esp_ble_mesh_cfg_model_pub_va_set_t::sub_addr (C++ member), 359
- esp_ble_mesh_cfg_model_sub_add_t (C++ class), 357
- esp_ble_mesh_cfg_model_sub_add_t::company_id (C++ member), 357
- esp_ble_mesh_cfg_model_sub_add_t::element_addr (C++ member), 357
- esp_ble_mesh_cfg_model_sub_add_t::model_id (C++ member), 357
- esp_ble_mesh_cfg_model_sub_add_t::sub_addr (C++ member), 357
- esp_ble_mesh_cfg_model_sub_delete_all_t (C++ class), 359
- esp_ble_mesh_cfg_model_sub_delete_all_t::company_id (C++ member), 359
- esp_ble_mesh_cfg_model_sub_delete_all_t::element_addr (C++ member), 359
- esp_ble_mesh_cfg_model_sub_delete_all_t::model_id (C++ member), 359
- esp_ble_mesh_cfg_model_sub_delete_all_t::sub_addr (C++ member), 359
- esp_ble_mesh_cfg_model_sub_delete_t (C++ class), 357
- esp_ble_mesh_cfg_model_sub_delete_t::company_id (C++ member), 357
- esp_ble_mesh_cfg_model_sub_delete_t::element_addr (C++ member), 357
- esp_ble_mesh_cfg_model_sub_delete_t::model_id (C++ member), 357
- esp_ble_mesh_cfg_model_sub_delete_t::sub_addr (C++ member), 357
- esp_ble_mesh_cfg_model_sub_list_cb_t (C++ class), 365
- esp_ble_mesh_cfg_model_sub_list_cb_t::company_id (C++ member), 365
- esp_ble_mesh_cfg_model_sub_list_cb_t::element_addr (C++ member), 365
- esp_ble_mesh_cfg_model_sub_list_cb_t::model_id (C++ member), 365
- esp_ble_mesh_cfg_model_sub_list_cb_t::status (C++ member), 365
- esp_ble_mesh_cfg_model_sub_list_cb_t::sub_addr (C++ member), 365
- esp_ble_mesh_cfg_model_sub_overwrite_t (C++ class), 357
- esp_ble_mesh_cfg_model_sub_overwrite_t::company_id (C++ member), 357
- esp_ble_mesh_cfg_model_sub_overwrite_t::element_addr (C++ member), 357
- esp_ble_mesh_cfg_model_sub_overwrite_t::model_id (C++ member), 357
- esp_ble_mesh_cfg_model_sub_overwrite_t::sub_addr (C++ member), 357
- esp_ble_mesh_cfg_model_sub_status_cb_t (C++ class), 363
- esp_ble_mesh_cfg_model_sub_status_cb_t::company_id (C++ member), 363
- esp_ble_mesh_cfg_model_sub_status_cb_t::element_addr (C++ member), 363
- esp_ble_mesh_cfg_model_sub_status_cb_t::model_id (C++ member), 363

- (*C macro*), 316
- ESP_BLE_MESH_CFG_STATUS_STORAGE_FAILURE (*C macro*), 316
- ESP_BLE_MESH_CFG_STATUS_SUCCESS (*C macro*), 316
- esp_ble_mesh_cfg_status_t (*C++ type*), 323
- ESP_BLE_MESH_CFG_STATUS_TEMP_UNABLE_TO_CHANGE_STATE (*C macro*), 316
- ESP_BLE_MESH_CFG_STATUS_UNSPECIFIED_ERROR (*C macro*), 316
- esp_ble_mesh_cfg_vnd_model_app_get_t (*C++ class*), 354
- esp_ble_mesh_cfg_vnd_model_app_get_t::company (*C++ member*), 354
- esp_ble_mesh_cfg_vnd_model_app_get_t::element (*C++ member*), 354
- esp_ble_mesh_cfg_vnd_model_app_get_t::model_id (*C++ member*), 354
- esp_ble_mesh_cfg_vnd_model_sub_get_t (*C++ class*), 353
- esp_ble_mesh_cfg_vnd_model_sub_get_t::company (*C++ member*), 354
- esp_ble_mesh_cfg_vnd_model_sub_get_t::element (*C++ member*), 354
- esp_ble_mesh_cfg_vnd_model_sub_get_t::model_id (*C++ member*), 354
- ESP_BLE_MESH_CID_NVAL (*C macro*), 308
- esp_ble_mesh_client_common_param_t (*C++ class*), 306
- esp_ble_mesh_client_common_param_t::ctx (*C++ member*), 306
- esp_ble_mesh_client_common_param_t::model_id (*C++ member*), 306
- esp_ble_mesh_client_common_param_t::msg_role (*C++ member*), 306
- esp_ble_mesh_client_common_param_t::msg_timeout (*C++ member*), 306
- esp_ble_mesh_client_common_param_t::opcode (*C++ member*), 306
- esp_ble_mesh_client_model_deinit (*C++ function*), 334
- esp_ble_mesh_client_model_init (*C++ function*), 334
- ESP_BLE_MESH_CLIENT_MODEL_RECV_PUBLISH_MSG_EVT (*C++ enumerator*), 330
- esp_ble_mesh_client_model_send_msg (*C++ function*), 335
- ESP_BLE_MESH_CLIENT_MODEL_SEND_TIMEOUT_EVT (*C++ enumerator*), 330
- esp_ble_mesh_client_op_pair_t (*C++ class*), 306
- esp_ble_mesh_client_op_pair_t::cli_op (*C++ member*), 306
- esp_ble_mesh_client_op_pair_t::status (*C++ member*), 306
- esp_ble_mesh_client_t (*C++ class*), 306
- esp_ble_mesh_client_t::internal_data (*C++ member*), 306
- esp_ble_mesh_client_t::model (*C++ member*), 306
- esp_ble_mesh_client_t::msg_role (*C++ member*), 306
- esp_ble_mesh_client_t::op_pair (*C++ member*), 306
- esp_ble_mesh_client_t::op_pair_size (*C++ member*), 306
- esp_ble_mesh_client_t::publish_status (*C++ member*), 306
- esp_ble_mesh_comp_t (*C++ class*), 303
- esp_ble_mesh_comp_t::cid (*C++ member*), 303
- esp_ble_mesh_comp_t::element_count (*C++ member*), 303
- esp_ble_mesh_comp_t::elements (*C++ member*), 303
- esp_ble_mesh_comp_t::pid (*C++ member*), 303
- esp_ble_mesh_comp_t::vid (*C++ member*), 303
- ESP_BLE_MESH_CONDENSATION_ERROR (*C macro*), 408
- ESP_BLE_MESH_CONDENSATION_WARNING (*C macro*), 408
- esp_ble_mesh_config_client_get_state (*C++ function*), 347
- esp_ble_mesh_config_client_set_state (*C++ function*), 348
- ESP_BLE_MESH_CONFIGURATION_ERROR (*C macro*), 408
- ESP_BLE_MESH_CONFIGURATION_WARNING (*C macro*), 408
- esp_ble_mesh_deinit (*C++ function*), 331
- ESP_BLE_MESH_DEINIT_MESH_COMP_EVT (*C++ enumerator*), 329
- esp_ble_mesh_deinit_param_t (*C++ class*), 300
- esp_ble_mesh_deinit_param_t::erase_flash (*C++ member*), 300
- esp_ble_mesh_dev_add_flag_t (*C++ type*), 323
- esp_ble_mesh_dev_role_t (*C++ enum*), 325
- esp_ble_mesh_device_delete_t (*C++ class*), 304
- esp_ble_mesh_device_delete_t::addr (*C++ member*), 304
- esp_ble_mesh_device_delete_t::addr_type (*C++ member*), 304
- esp_ble_mesh_device_delete_t::flag (*C++ member*), 304
- esp_ble_mesh_device_delete_t::uuid (*C++ member*), 304
- ESP_BLE_MESH_DEVICE_DROPPED_ERROR (*C macro*), 409
- ESP_BLE_MESH_DEVICE_DROPPED_WARNING (*C macro*), 409
- ESP_BLE_MESH_DEVICE_MOVED_ERROR (*C macro*), 409

- macro*), 409
- ESP_BLE_MESH_DEVICE_MOVED_WARNING (C *macro*), 409
- ESP_BLE_MESH_DEVICE_NAME_MAX_LEN (C *macro*), 308
- ESP_BLE_MESH_DISPLAY_NUMBER (C++ *enumerator*), 324
- ESP_BLE_MESH_DISPLAY_STRING (C++ *enumerator*), 324
- esp_ble_mesh_elem_t (C++ *class*), 300
- esp_ble_mesh_elem_t::element_addr (C++ *member*), 300
- esp_ble_mesh_elem_t::location (C++ *member*), 300
- esp_ble_mesh_elem_t::sig_model_count (C++ *member*), 300
- esp_ble_mesh_elem_t::sig_models (C++ *member*), 301
- esp_ble_mesh_elem_t::vnd_model_count (C++ *member*), 300
- esp_ble_mesh_elem_t::vnd_models (C++ *member*), 301
- ESP_BLE_MESH_ELEMENT (C *macro*), 311
- ESP_BLE_MESH_ELEMENT_NOT_CALIBRATED_ERROR (C *macro*), 408
- ESP_BLE_MESH_ELEMENT_NOT_CALIBRATED_WARNING (C *macro*), 408
- ESP_BLE_MESH_EMPTY_ERROR (C *macro*), 409
- ESP_BLE_MESH_EMPTY_WARNING (C *macro*), 409
- ESP_BLE_MESH_ENTER_NUMBER (C++ *enumerator*), 325
- ESP_BLE_MESH_ENTER_STRING (C++ *enumerator*), 325
- esp_ble_mesh_fast_prov_action_t (C++ *enum*), 325
- esp_ble_mesh_fast_prov_info_t (C++ *class*), 305
- esp_ble_mesh_fast_prov_info_t::flags (C++ *member*), 305
- esp_ble_mesh_fast_prov_info_t::iv_index (C++ *member*), 305
- esp_ble_mesh_fast_prov_info_t::match_len (C++ *member*), 305
- esp_ble_mesh_fast_prov_info_t::match_val (C++ *member*), 305
- esp_ble_mesh_fast_prov_info_t::net_idx (C++ *member*), 305
- esp_ble_mesh_fast_prov_info_t::offset (C++ *member*), 305
- esp_ble_mesh_fast_prov_info_t::unicast_max (C++ *member*), 305
- esp_ble_mesh_fast_prov_info_t::unicast_min (C++ *member*), 305
- ESP_BLE_MESH_FEATURE_ALL_SUPPORTED (C *macro*), 309
- ESP_BLE_MESH_FEATURE_FRIEND (C *macro*), 309
- ESP_BLE_MESH_FEATURE_LOW_POWER (C *macro*), 309
- ESP_BLE_MESH_FEATURE_PROXY (C *macro*), 309
- ESP_BLE_MESH_FEATURE_RELAY (C *macro*), 309
- esp_ble_mesh_find_element (C++ *function*), 331
- esp_ble_mesh_find_sig_model (C++ *function*), 332
- esp_ble_mesh_find_vendor_model (C++ *function*), 332
- ESP_BLE_MESH_FRIEND_DISABLED (C *macro*), 309
- ESP_BLE_MESH_FRIEND_ENABLED (C *macro*), 309
- ESP_BLE_MESH_FRIEND_FRIENDSHIP_ESTABLISH_EVT (C++ *enumerator*), 328
- ESP_BLE_MESH_FRIEND_FRIENDSHIP_TERMINATE_EVT (C++ *enumerator*), 328
- ESP_BLE_MESH_FRIEND_NOT_SUPPORTED (C *macro*), 309
- ESP_BLE_MESH_GATT_PROXY_DISABLED (C *macro*), 309
- ESP_BLE_MESH_GATT_PROXY_ENABLED (C *macro*), 309
- ESP_BLE_MESH_GATT_PROXY_NOT_SUPPORTED (C *macro*), 309
- ESP_BLE_MESH_GEN_ADMIN_ACCESS_READ (C++ *enumerator*), 401
- ESP_BLE_MESH_GEN_ADMIN_ACCESS_READ_WRITE (C++ *enumerator*), 401
- ESP_BLE_MESH_GEN_ADMIN_ACCESS_WRITE (C++ *enumerator*), 401
- ESP_BLE_MESH_GEN_ADMIN_NOT_USER_PROP (C++ *enumerator*), 401
- esp_ble_mesh_gen_admin_prop_access_t (C++ *enum*), 401
- esp_ble_mesh_gen_admin_prop_srv_t (C++ *class*), 389
- esp_ble_mesh_gen_admin_prop_srv_t::model (C++ *member*), 389
- esp_ble_mesh_gen_admin_prop_srv_t::properties (C++ *member*), 389
- esp_ble_mesh_gen_admin_prop_srv_t::property_count (C++ *member*), 389
- esp_ble_mesh_gen_admin_prop_srv_t::rsp_ctrl (C++ *member*), 389
- esp_ble_mesh_gen_admin_properties_status_cb_t (C++ *class*), 382
- esp_ble_mesh_gen_admin_properties_status_cb_t::pr (C++ *member*), 383
- esp_ble_mesh_gen_admin_property_get_t (C++ *class*), 379
- esp_ble_mesh_gen_admin_property_get_t::property_i (C++ *member*), 379
- esp_ble_mesh_gen_admin_property_set_t (C++ *class*), 379
- esp_ble_mesh_gen_admin_property_set_t::property_i (C++ *member*), 379
- esp_ble_mesh_gen_admin_property_set_t::property_v

(C++ member), 379
 esp_ble_mesh_gen_admin_property_set_t: ~~assembly~~ esp_ble_mesh_gen_client_status_cb_t
 (C++ member), 379 (C++ union), 373
 esp_ble_mesh_gen_admin_property_status_cb_t: ~~assembly~~ esp_ble_mesh_gen_client_status_cb_t::admin_proper
 (C++ class), 383 (C++ member), 373
 esp_ble_mesh_gen_admin_property_status_cb_t: ~~assembly~~ esp_ble_mesh_gen_client_status_cb_t::admin_proper
 (C++ member), 383 (C++ member), 373
 esp_ble_mesh_gen_admin_property_status_cb_t: ~~assembly~~ esp_ble_mesh_gen_client_status_cb_t::battery_stat
 (C++ member), 383 (C++ member), 373
 esp_ble_mesh_gen_admin_property_status_cb_t: ~~assembly~~ esp_ble_mesh_gen_client_status_cb_t::client_prope
 (C++ member), 383 (C++ member), 374
 esp_ble_mesh_gen_admin_property_status_cb_t: ~~assembly~~ esp_ble_mesh_gen_client_status_cb_t::def_trans_ti
 (C++ member), 383 (C++ member), 373
 esp_ble_mesh_gen_battery_srv_t (C++ esp_ble_mesh_gen_client_status_cb_t::level_status
 class), 387 (C++ member), 373
 esp_ble_mesh_gen_battery_srv_t::model esp_ble_mesh_gen_client_status_cb_t::location_glo
 (C++ member), 387 (C++ member), 373
 esp_ble_mesh_gen_battery_srv_t::rsp_ctrl esp_ble_mesh_gen_client_status_cb_t::location_loc
 (C++ member), 387 (C++ member), 373
 esp_ble_mesh_gen_battery_srv_t::state esp_ble_mesh_gen_client_status_cb_t::manufacturer
 (C++ member), 387 (C++ member), 374
 esp_ble_mesh_gen_battery_state_t (C++ esp_ble_mesh_gen_client_status_cb_t::manufacturer
 class), 387 (C++ member), 374
 esp_ble_mesh_gen_battery_state_t::battery esp_ble_mesh_gen_client_status_cb_t::onoff_status
 (C++ member), 387 (C++ member), 373
 esp_ble_mesh_gen_battery_state_t::battery esp_ble_mesh_gen_client_status_cb_t::onpowerup_st
 (C++ member), 387 (C++ member), 373
 esp_ble_mesh_gen_battery_state_t::time esp_ble_mesh_gen_client_status_cb_t::power_defaul
 (C++ member), 387 (C++ member), 373
 esp_ble_mesh_gen_battery_state_t::time esp_ble_mesh_gen_client_status_cb_t::power_last_s
 (C++ member), 387 (C++ member), 373
 esp_ble_mesh_gen_battery_status_cb_t esp_ble_mesh_gen_client_status_cb_t::power_level_
 (C++ class), 381 (C++ member), 373
 esp_ble_mesh_gen_battery_status_cb_t::battery esp_ble_mesh_gen_client_status_cb_t::power_range_
 (C++ member), 381 (C++ member), 373
 esp_ble_mesh_gen_battery_status_cb_t::flags esp_ble_mesh_gen_client_status_cb_t::user_proper
 (C++ member), 381 (C++ member), 373
 esp_ble_mesh_gen_battery_status_cb_t::time esp_ble_mesh_gen_client_status_cb_t::user_proper
 (C++ member), 381 (C++ member), 373
 esp_ble_mesh_gen_battery_status_cb_t::time esp_ble_mesh_gen_def_trans_time_set_t
 (C++ member), 381 (C++ class), 377
 esp_ble_mesh_gen_client_prop_srv_t esp_ble_mesh_gen_def_trans_time_set_t::trans_time
 (C++ class), 389 (C++ member), 377
 esp_ble_mesh_gen_client_prop_srv_t::id esp_ble_mesh_gen_def_trans_time_srv_t
 (C++ member), 390 (C++ class), 385
 esp_ble_mesh_gen_client_prop_srv_t::model esp_ble_mesh_gen_def_trans_time_srv_t::model
 (C++ member), 390 (C++ member), 385
 esp_ble_mesh_gen_client_prop_srv_t::prop_style esp_ble_mesh_gen_def_trans_time_srv_t::rsp_ctrl
 (C++ member), 390 (C++ member), 385
 esp_ble_mesh_gen_client_prop_srv_t::rsp_ctrl esp_ble_mesh_gen_def_trans_time_srv_t::state
 (C++ member), 390 (C++ member), 385
 esp_ble_mesh_gen_client_properties_get esp_ble_mesh_gen_def_trans_time_state_t
 (C++ class), 379 (C++ class), 385
 esp_ble_mesh_gen_client_properties_get esp_ble_mesh_gen_def_trans_time_state_t::trans_ti
 (C++ member), 379 (C++ member), 385
 esp_ble_mesh_gen_client_properties_status_cb_t esp_ble_mesh_gen_def_trans_time_status_cb_t
 (C++ class), 383 (C++ class), 380
 esp_ble_mesh_gen_client_properties_status_cb_t: ~~assembly~~ esp_ble_mesh_gen_def_trans_time_status_cb_t::tran

- (C++ member), 380
- `esp_ble_mesh_gen_delta_set_t` (C++ class), 376
- `esp_ble_mesh_gen_delta_set_t::delay` (C++ member), 376
- `esp_ble_mesh_gen_delta_set_t::level` (C++ member), 376
- `esp_ble_mesh_gen_delta_set_t::op_en` (C++ member), 376
- `esp_ble_mesh_gen_delta_set_t::tid` (C++ member), 376
- `esp_ble_mesh_gen_delta_set_t::trans_time` (C++ member), 376
- `esp_ble_mesh_gen_level_set_t` (C++ class), 376
- `esp_ble_mesh_gen_level_set_t::delay` (C++ member), 376
- `esp_ble_mesh_gen_level_set_t::level` (C++ member), 376
- `esp_ble_mesh_gen_level_set_t::op_en` (C++ member), 376
- `esp_ble_mesh_gen_level_set_t::tid` (C++ member), 376
- `esp_ble_mesh_gen_level_set_t::trans_time` (C++ member), 376
- `esp_ble_mesh_gen_level_srv_t` (C++ class), 385
- `esp_ble_mesh_gen_level_srv_t::last` (C++ member), 385
- `esp_ble_mesh_gen_level_srv_t::model` (C++ member), 385
- `esp_ble_mesh_gen_level_srv_t::rsp_ctrl` (C++ member), 385
- `esp_ble_mesh_gen_level_srv_t::state` (C++ member), 385
- `esp_ble_mesh_gen_level_srv_t::transition` (C++ member), 385
- `esp_ble_mesh_gen_level_srv_t::tt_delta_srv` (C++ member), 385
- `esp_ble_mesh_gen_level_state_t` (C++ class), 384
- `esp_ble_mesh_gen_level_state_t::last_delta` (C++ member), 384
- `esp_ble_mesh_gen_level_state_t::last_level` (C++ member), 384
- `esp_ble_mesh_gen_level_state_t::level` (C++ member), 384
- `esp_ble_mesh_gen_level_state_t::move_step` (C++ member), 384
- `esp_ble_mesh_gen_level_state_t::positive` (C++ member), 385
- `esp_ble_mesh_gen_level_state_t::target_level` (C++ member), 384
- `esp_ble_mesh_gen_level_status_cb_t` (C++ class), 380
- `esp_ble_mesh_gen_level_status_cb_t::op_en` (C++ member), 380
- `esp_ble_mesh_gen_level_status_cb_t::presplit_level` (C++ member), 380
- `esp_ble_mesh_gen_level_status_cb_t::remain_time` (C++ member), 380
- `esp_ble_mesh_gen_level_status_cb_t::target_level` (C++ member), 380
- `esp_ble_mesh_gen_loc_global_set_t` (C++ class), 378
- `esp_ble_mesh_gen_loc_global_set_t::global_altitude` (C++ member), 378
- `esp_ble_mesh_gen_loc_global_set_t::global_latitude` (C++ member), 378
- `esp_ble_mesh_gen_loc_global_set_t::global_longitude` (C++ member), 378
- `esp_ble_mesh_gen_loc_global_status_cb_t` (C++ class), 381
- `esp_ble_mesh_gen_loc_global_status_cb_t::global_altitude` (C++ member), 382
- `esp_ble_mesh_gen_loc_global_status_cb_t::global_latitude` (C++ member), 382
- `esp_ble_mesh_gen_loc_global_status_cb_t::global_longitude` (C++ member), 382
- `esp_ble_mesh_gen_loc_local_set_t` (C++ class), 378
- `esp_ble_mesh_gen_loc_local_set_t::floor_number` (C++ member), 378
- `esp_ble_mesh_gen_loc_local_set_t::local_altitude` (C++ member), 378
- `esp_ble_mesh_gen_loc_local_set_t::local_east` (C++ member), 378
- `esp_ble_mesh_gen_loc_local_set_t::local_north` (C++ member), 378
- `esp_ble_mesh_gen_loc_local_set_t::uncertainty` (C++ member), 378
- `esp_ble_mesh_gen_loc_local_status_cb_t` (C++ class), 382
- `esp_ble_mesh_gen_loc_local_status_cb_t::floor_number` (C++ member), 382
- `esp_ble_mesh_gen_loc_local_status_cb_t::local_altitude` (C++ member), 382
- `esp_ble_mesh_gen_loc_local_status_cb_t::local_east` (C++ member), 382
- `esp_ble_mesh_gen_loc_local_status_cb_t::local_north` (C++ member), 382
- `esp_ble_mesh_gen_loc_local_status_cb_t::uncertainty` (C++ member), 382
- `esp_ble_mesh_gen_location_setup_srv_t` (C++ class), 388
- `esp_ble_mesh_gen_location_setup_srv_t::model` (C++ member), 388
- `esp_ble_mesh_gen_location_setup_srv_t::rsp_ctrl` (C++ member), 388
- `esp_ble_mesh_gen_location_setup_srv_t::state` (C++ member), 388
- `esp_ble_mesh_gen_location_srv_t` (C++ class), 388
- `esp_ble_mesh_gen_location_srv_t::model` (C++ member), 388
- `esp_ble_mesh_gen_location_srv_t::rsp_ctrl` (C++ member), 388

(C++ member), 377
 esp_ble_mesh_gen_onpowerup_state_t (C++ class), 385
 esp_ble_mesh_gen_onpowerup_state_t::onpowerup (C++ member), 385
 esp_ble_mesh_gen_onpowerup_status_cb_t (C++ class), 380
 esp_ble_mesh_gen_onpowerup_status_cb_t::onpowerup (C++ member), 380
 esp_ble_mesh_gen_power_default_set_t (C++ class), 377
 esp_ble_mesh_gen_power_default_set_t::power (C++ member), 378
 esp_ble_mesh_gen_power_default_status_cb_t (C++ class), 381
 esp_ble_mesh_gen_power_default_status_cb_t::onpowerup (C++ member), 381
 esp_ble_mesh_gen_power_last_status_cb_t (C++ class), 381
 esp_ble_mesh_gen_power_last_status_cb_t::remain (C++ member), 381
 esp_ble_mesh_gen_power_level_set_t (C++ class), 377
 esp_ble_mesh_gen_power_level_set_t::delete (C++ member), 377
 esp_ble_mesh_gen_power_level_set_t::open (C++ member), 377
 esp_ble_mesh_gen_power_level_set_t::power (C++ member), 377
 esp_ble_mesh_gen_power_level_set_t::time (C++ member), 377
 esp_ble_mesh_gen_power_level_set_t::transition (C++ member), 377
 esp_ble_mesh_gen_power_level_set_t::transition_time (C++ member), 377
 esp_ble_mesh_gen_power_level_setup_srv_t (C++ class), 387
 esp_ble_mesh_gen_power_level_setup_srv_t::model (C++ member), 387
 esp_ble_mesh_gen_power_level_setup_srv_t::onpowerup (C++ member), 387
 esp_ble_mesh_gen_power_level_setup_srv_t::onpowerup_cancel (C++ member), 387
 esp_ble_mesh_gen_power_level_setup_srv_t::state (C++ member), 387
 esp_ble_mesh_gen_power_level_srv_t (C++ class), 386
 esp_ble_mesh_gen_power_level_srv_t::last (C++ member), 387
 esp_ble_mesh_gen_power_level_srv_t::model (C++ member), 386
 esp_ble_mesh_gen_power_level_srv_t::onpowerup (C++ member), 386
 esp_ble_mesh_gen_power_level_srv_t::state (C++ member), 386
 esp_ble_mesh_gen_power_level_srv_t::transition (C++ member), 387
 esp_ble_mesh_gen_power_level_srv_t::transition_time (C++ member), 387
 esp_ble_mesh_gen_power_level_srv_t::tt (C++ member), 387
 esp_ble_mesh_gen_power_level_state_t (C++ class), 386
 esp_ble_mesh_gen_power_level_state_t::power (C++ member), 386
 (C++ member), 386
 esp_ble_mesh_gen_power_level_state_t::power_default (C++ member), 386
 esp_ble_mesh_gen_power_level_state_t::power_last (C++ member), 386
 esp_ble_mesh_gen_power_level_state_t::power_range (C++ member), 386
 esp_ble_mesh_gen_power_level_state_t::power_range_min (C++ member), 386
 esp_ble_mesh_gen_power_level_state_t::power_range_max (C++ member), 386
 esp_ble_mesh_gen_power_level_state_t::status_code (C++ member), 386
 esp_ble_mesh_gen_power_level_state_t::target_power (C++ member), 386
 esp_ble_mesh_gen_power_level_status_cb_t (C++ class), 380
 esp_ble_mesh_gen_power_level_status_cb_t::open (C++ member), 380
 esp_ble_mesh_gen_power_level_status_cb_t::present (C++ member), 380
 esp_ble_mesh_gen_power_level_status_cb_t::remain (C++ member), 381
 esp_ble_mesh_gen_power_level_status_cb_t::target (C++ member), 381
 esp_ble_mesh_gen_power_onoff_setup_srv_t (C++ class), 386
 esp_ble_mesh_gen_power_onoff_setup_srv_t::model (C++ member), 386
 esp_ble_mesh_gen_power_onoff_setup_srv_t::rsp_ctrl (C++ member), 386
 esp_ble_mesh_gen_power_onoff_setup_srv_t::state (C++ member), 386
 esp_ble_mesh_gen_power_onoff_srv_t (C++ class), 385
 esp_ble_mesh_gen_power_onoff_srv_t::model (C++ member), 386
 esp_ble_mesh_gen_power_onoff_srv_t::rsp_ctrl (C++ member), 386
 esp_ble_mesh_gen_power_onoff_srv_t::state (C++ member), 386
 esp_ble_mesh_gen_power_range_set_t (C++ class), 378
 esp_ble_mesh_gen_power_range_set_t::range_max (C++ member), 378
 esp_ble_mesh_gen_power_range_set_t::range_min (C++ member), 378
 esp_ble_mesh_gen_power_range_status_cb_t (C++ class), 381
 esp_ble_mesh_gen_power_range_status_cb_t::range_max (C++ member), 381
 esp_ble_mesh_gen_power_range_status_cb_t::range_min (C++ member), 381
 esp_ble_mesh_gen_power_range_status_cb_t::status (C++ member), 381
 ESP_BLE_MESH_GEN_USER_ACCESS_PROHIBIT (C++ enumerator), 401
 ESP_BLE_MESH_GEN_USER_ACCESS_READ (C++ enumerator), 401
 ESP_BLE_MESH_GEN_USER_ACCESS_READ_WRITE (C++ enumerator), 401

class), 388
 esp_ble_mesh_generic_property_t::admin_accessible (C++ member), 389
 esp_ble_mesh_generic_property_t::id (C++ member), 388
 esp_ble_mesh_generic_property_t::manual_accessible (C++ member), 389
 esp_ble_mesh_generic_property_t::user_accessible (C++ member), 388
 esp_ble_mesh_generic_property_t::value (C++ member), 389
 esp_ble_mesh_generic_server_cb_event_t (C++ enum), 401
 esp_ble_mesh_generic_server_cb_param_t (C++ class), 396
 esp_ble_mesh_generic_server_cb_param_t::enable (C++ member), 396
 esp_ble_mesh_generic_server_cb_param_t::enable_model (C++ member), 396
 esp_ble_mesh_generic_server_cb_param_t::ESP_BLE_MESH_GENERIC_SERVER_STATE_CHANGE_EVT (C++ member), 396
 esp_ble_mesh_generic_server_cb_t (C++ type), 400
 esp_ble_mesh_generic_server_cb_value_t (C++ union), 375
 esp_ble_mesh_generic_server_cb_value_t::enable (C++ member), 375
 esp_ble_mesh_generic_server_cb_value_t::enable_model (C++ member), 375
 esp_ble_mesh_generic_server_cb_value_t::enable_model_change (C++ member), 375
 ESP_BLE_MESH_GENERIC_SERVER_EVT_MAX (C++ enumerator), 402
 ESP_BLE_MESH_GENERIC_SERVER_RECV_GET_MSG_EVT (C++ enumerator), 401
 esp_ble_mesh_generic_server_rcv_get_msg_evt (C++ union), 374
 esp_ble_mesh_generic_server_rcv_get_msg_evt::admin_accessible (C++ member), 375
 esp_ble_mesh_generic_server_rcv_get_msg_evt::id (C++ member), 375
 esp_ble_mesh_generic_server_rcv_get_msg_evt::manual_accessible (C++ member), 375
 esp_ble_mesh_generic_server_rcv_get_msg_evt::user_accessible (C++ member), 374
 ESP_BLE_MESH_GENERIC_SERVER_RECV_SET_MSG_EVT (C++ enumerator), 401
 esp_ble_mesh_generic_server_rcv_set_msg_evt (C++ union), 375
 esp_ble_mesh_generic_server_rcv_set_msg_evt::admin_accessible (C++ member), 375
 esp_ble_mesh_generic_server_rcv_set_msg_evt::id (C++ member), 375
 esp_ble_mesh_generic_server_rcv_set_msg_evt::manual_accessible (C++ member), 375
 esp_ble_mesh_generic_server_rcv_set_msg_evt::user_accessible (C++ member), 374
 esp_ble_mesh_generic_server_rcv_set_msg_evt::update_mesh_get_element_count (C++ member), 375
 esp_ble_mesh_generic_server_rcv_set_msg_evt::update_mesh_get_fast_prov_app_key (C++ member), 375
 esp_ble_mesh_generic_server_rcv_set_msg_evt::update_mesh_get_model_publish_period (C++ member), 375

(C++ function), 331
 esp_ble_mesh_get_primary_element_address (C++ function), 331
 ESP_BLE_MESH_GET_PUBLISH_TRANSMIT_COUNT (C macro), 310
 ESP_BLE_MESH_GET_PUBLISH_TRANSMIT_INTERVAL (C macro), 310
 ESP_BLE_MESH_GET_SENSOR_DATA_FORMAT (C macro), 465
 ESP_BLE_MESH_GET_SENSOR_DATA_LENGTH (C macro), 465
 ESP_BLE_MESH_GET_SENSOR_DATA_PROPERTY_ID (C macro), 465
 ESP_BLE_MESH_GET_TRANSMIT_COUNT (C macro), 310
 ESP_BLE_MESH_GET_TRANSMIT_INTERVAL (C macro), 310
 esp_ble_mesh_health_attention_off_cb_t (C++ class), 407
 esp_ble_mesh_health_attention_off_cb_t::model (C++ member), 407
 esp_ble_mesh_health_attention_on_cb_t (C++ class), 407
 esp_ble_mesh_health_attention_on_cb_t::model (C++ member), 407
 esp_ble_mesh_health_attention_on_cb_t::ESP_BLE_MESH_HEALTH_CLIENT_TIMEOUT_EVT (C++ member), 407
 esp_ble_mesh_health_attention_set_t (C++ class), 405
 esp_ble_mesh_health_attention_set_t::attention (C++ member), 405
 esp_ble_mesh_health_attention_status_cb_t (C++ class), 406
 esp_ble_mesh_health_attention_status_cb_t::test_id (C++ member), 406
 esp_ble_mesh_health_client_cb_event_t (C++ enum), 410
 esp_ble_mesh_health_client_cb_param_t (C++ class), 406
 esp_ble_mesh_health_client_cb_param_t::company_id (C++ member), 406
 esp_ble_mesh_health_client_cb_param_t::params (C++ member), 406
 esp_ble_mesh_health_client_cb_param_t::status (C++ member), 406
 esp_ble_mesh_health_client_cb_t (C++ type), 409
 esp_ble_mesh_health_client_common_cb_param_t (C++ union), 403
 esp_ble_mesh_health_client_common_cb_param_t::attention_status (C++ member), 403
 esp_ble_mesh_health_client_common_cb_param_t::over_the_air_status (C++ member), 403
 esp_ble_mesh_health_client_common_cb_param_t::fault_status (C++ member), 403
 esp_ble_mesh_health_client_common_cb_param_t::period_status (C++ member), 403
 ESP_BLE_MESH_HEALTH_CLIENT_EVT_MAX (C++ enumerator), 410
 esp_ble_mesh_health_client_get_state (C++ function), 402
 ESP_BLE_MESH_HEALTH_CLIENT_GET_STATE_EVT (C++ enumerator), 410
 esp_ble_mesh_health_client_get_state_t (C++ union), 403
 esp_ble_mesh_health_client_get_state_t::fault_get (C++ member), 403
 ESP_BLE_MESH_HEALTH_CLIENT_PUBLISH_EVT (C++ enumerator), 410
 esp_ble_mesh_health_client_set_state (C++ function), 402
 ESP_BLE_MESH_HEALTH_CLIENT_SET_STATE_EVT (C++ enumerator), 410
 esp_ble_mesh_health_client_set_state_t (C++ union), 403
 esp_ble_mesh_health_client_set_state_t::attention (C++ member), 403
 esp_ble_mesh_health_client_set_state_t::fault_clear (C++ member), 403
 esp_ble_mesh_health_client_set_state_t::fault_test (C++ member), 403
 esp_ble_mesh_health_client_set_state_t::period_set (C++ member), 403
 ESP_BLE_MESH_HEALTH_CLIENT_TIMEOUT_EVT (C++ enumerator), 410
 esp_ble_mesh_health_current_status_cb_t (C++ class), 405
 esp_ble_mesh_health_current_status_cb_t::company_id (C++ member), 406
 esp_ble_mesh_health_current_status_cb_t::fault_array (C++ member), 406
 esp_ble_mesh_health_current_status_cb_t::test_id (C++ member), 406
 ESP_BLE_MESH_HEALTH_FAULT_ARRAY_SIZE (C macro), 409
 esp_ble_mesh_health_fault_clear_cb_t (C++ class), 407
 esp_ble_mesh_health_fault_clear_cb_t::company_id (C++ member), 407
 esp_ble_mesh_health_fault_clear_cb_t::model (C++ member), 407
 esp_ble_mesh_health_fault_clear_t (C++ class), 405
 esp_ble_mesh_health_fault_clear_t::company_id (C++ member), 405
 esp_ble_mesh_health_fault_get_t (C++ class), 405
 esp_ble_mesh_health_fault_get_t::company_id (C++ member), 405
 esp_ble_mesh_health_fault_status_cb_t (C++ class), 406
 esp_ble_mesh_health_fault_status_cb_t::company_id (C++ member), 406
 esp_ble_mesh_health_fault_status_cb_t::fault_array (C++ member), 406
 esp_ble_mesh_health_fault_status_cb_t::test_id (C++ member), 406

- (C++ member), 406
- esp_ble_mesh_health_fault_test_cb_t (C++ class), 407
- esp_ble_mesh_health_fault_test_cb_t::company_id (C++ member), 407
- esp_ble_mesh_health_fault_test_cb_t::model_id (C++ member), 407
- esp_ble_mesh_health_fault_test_cb_t::test_id (C++ member), 407
- esp_ble_mesh_health_fault_test_t (C++ class), 405
- esp_ble_mesh_health_fault_test_t::company_id (C++ member), 405
- esp_ble_mesh_health_fault_test_t::test_id (C++ member), 405
- esp_ble_mesh_health_fault_update_comp_cb_t (C++ class), 406
- esp_ble_mesh_health_fault_update_comp_cb_t::event (C++ member), 407
- esp_ble_mesh_health_fault_update_comp_cb_t::event_id (C++ member), 407
- esp_ble_mesh_health_model_status_t (C++ type), 323
- esp_ble_mesh_health_period_set_t (C++ class), 405
- esp_ble_mesh_health_period_set_t::fast_period (C++ member), 405
- esp_ble_mesh_health_period_status_cb_t (C++ class), 406
- esp_ble_mesh_health_period_status_cb_t::fast_period_divisor (C++ member), 406
- ESP_BLE_MESH_HEALTH_PUB_DEFINE (C macro), 408
- ESP_BLE_MESH_HEALTH_SERVER_ATTENTION_OFF_EVT (C++ enumerator), 410
- ESP_BLE_MESH_HEALTH_SERVER_ATTENTION_ON_EVT (C++ enumerator), 410
- esp_ble_mesh_health_server_cb_event_t (C++ enum), 410
- esp_ble_mesh_health_server_cb_param_t (C++ union), 403
- esp_ble_mesh_health_server_cb_param_t::attention (C++ member), 404
- esp_ble_mesh_health_server_cb_param_t::attention_filter (C++ member), 404
- esp_ble_mesh_health_server_cb_param_t::fault (C++ member), 403
- esp_ble_mesh_health_server_cb_param_t::fault_clear (C++ member), 404
- esp_ble_mesh_health_server_cb_param_t::fault_test (C++ member), 403
- esp_ble_mesh_health_server_cb_t (C++ type), 409
- ESP_BLE_MESH_HEALTH_SERVER_EVT_MAX (C++ enumerator), 410
- ESP_BLE_MESH_HEALTH_SERVER_FAULT_CLEAR_EVT (C++ enumerator), 410
- ESP_BLE_MESH_HEALTH_SERVER_FAULT_TEST_EVT (C++ enumerator), 328
- (C++ enumerator), 410
- esp_ble_mesh_health_server_fault_update (C++ function), 402
- ESP_BLE_MESH_HEALTH_SERVER_FAULT_UPDATE_COMP_EVT (C++ enumerator), 410
- esp_ble_mesh_health_srv_cb_t (C++ class), 404
- esp_ble_mesh_health_srv_cb_t::attention_off (C++ member), 404
- esp_ble_mesh_health_srv_cb_t::attention_on (C++ member), 404
- esp_ble_mesh_health_srv_cb_t::fault_clear (C++ member), 404
- esp_ble_mesh_health_srv_cb_t::fault_test (C++ member), 404
- esp_ble_mesh_health_srv_t (C++ class), 404
- esp_ble_mesh_health_srv_t::attention_timer (C++ member), 404
- esp_ble_mesh_health_srv_t::attention_timer_start (C++ member), 405
- esp_ble_mesh_health_srv_t::health_cb (C++ member), 404
- esp_ble_mesh_health_srv_t::health_test (C++ member), 405
- esp_ble_mesh_health_srv_t::model (C++ member), 404
- ESP_BLE_MESH_HEALTH_STANDARD_TEST (C macro), 408
- esp_ble_mesh_health_test_t (C++ class), 404
- esp_ble_mesh_health_test_t::company_id (C++ member), 404
- esp_ble_mesh_health_test_t::current_faults (C++ member), 404
- esp_ble_mesh_health_test_t::id_count (C++ member), 404
- esp_ble_mesh_health_test_t::prev_test_id (C++ member), 404
- esp_ble_mesh_health_test_t::registered_faults (C++ member), 404
- esp_ble_mesh_health_test_t::test_ids (C++ member), 404
- ESP_BLE_MESH_HEARTBEAT_FILTER_ACCEPTLIST (C macro), 311
- ESP_BLE_MESH_HEARTBEAT_FILTER_ADD (C macro), 312
- esp_ble_mesh_heartbeat_filter_info_t (C++ class), 305
- esp_ble_mesh_heartbeat_filter_info_t::hb_dst (C++ member), 306
- esp_ble_mesh_heartbeat_filter_info_t::hb_src (C++ member), 306
- ESP_BLE_MESH_HEARTBEAT_FILTER_REJECTLIST (C macro), 312
- ESP_BLE_MESH_HEARTBEAT_FILTER_REMOVE (C macro), 312
- ESP_BLE_MESH_HEARTBEAT_MESSAGE_RECV_EVT (C++ enumerator), 328

- ESP_BLE_MESH_HOUSING_OPENED_ERROR (C macro), 409
- ESP_BLE_MESH_HOUSING_OPENED_WARNING (C macro), 409
- esp_ble_mesh_init (C++ function), 331
- esp_ble_mesh_input_action_t (C++ enum), 324
- ESP_BLE_MESH_INPUT_NO_CHANGE_ERROR (C macro), 409
- ESP_BLE_MESH_INPUT_NO_CHANGE_WARNING (C macro), 409
- ESP_BLE_MESH_INPUT_OOB (C++ enumerator), 324
- ESP_BLE_MESH_INPUT_TOO_HIGH_ERROR (C macro), 409
- ESP_BLE_MESH_INPUT_TOO_HIGH_WARNING (C macro), 409
- ESP_BLE_MESH_INPUT_TOO_LOW_ERROR (C macro), 409
- ESP_BLE_MESH_INPUT_TOO_LOW_WARNING (C macro), 409
- ESP_BLE_MESH_INTERNAL_BUS_ERROR (C macro), 409
- ESP_BLE_MESH_INTERNAL_BUS_WARNING (C macro), 409
- ESP_BLE_MESH_INVALID_NODE_INDEX (C macro), 309
- ESP_BLE_MESH_INVALID_SCENE_NUMBER (C macro), 486
- ESP_BLE_MESH_INVALID_SENSOR_PROPERTY_ID (C macro), 464
- ESP_BLE_MESH_INVALID_SENSOR_SETTING_PROPERTY_ID (C macro), 464
- ESP_BLE_MESH_INVALID_SETTINGS_IDX (C macro), 308
- esp_ble_mesh_is_model_subscribed_to_group (C++ function), 331
- ESP_BLE_MESH_KEY_ANY (C macro), 309
- ESP_BLE_MESH_KEY_DEV (C macro), 309
- ESP_BLE_MESH_KEY_PRIMARY (C macro), 309
- ESP_BLE_MESH_KEY_UNUSED (C macro), 309
- esp_ble_mesh_last_msg_info_t (C++ class), 307
- esp_ble_mesh_last_msg_info_t::dst (C++ member), 307
- esp_ble_mesh_last_msg_info_t::src (C++ member), 307
- esp_ble_mesh_last_msg_info_t::tid (C++ member), 307
- esp_ble_mesh_last_msg_info_t::timestamp (C++ member), 307
- ESP_BLE_MESH_LC_FADE (C++ enumerator), 451
- ESP_BLE_MESH_LC_FADE_ON (C++ enumerator), 451
- ESP_BLE_MESH_LC_FADE_STANDBY_AUTO (C++ enumerator), 451
- ESP_BLE_MESH_LC_FADE_STANDBY_MANUAL (C++ enumerator), 451
- ESP_BLE_MESH_LC_OFF (C++ enumerator), 451
- ESP_BLE_MESH_LC_PROLONG (C++ enumerator), 451
- ESP_BLE_MESH_LC_RUN (C++ enumerator), 451
- ESP_BLE_MESH_LC_STANDBY (C++ enumerator), 451
- esp_ble_mesh_lc_state_t (C++ enum), 451
- esp_ble_mesh_light_client_cb_event_t (C++ enum), 451
- esp_ble_mesh_light_client_cb_param_t (C++ class), 427
- esp_ble_mesh_light_client_cb_param_t::error_code (C++ member), 428
- esp_ble_mesh_light_client_cb_param_t::params (C++ member), 428
- esp_ble_mesh_light_client_cb_param_t::status_cb (C++ member), 428
- esp_ble_mesh_light_client_cb_t (C++ type), 451
- ESP_BLE_MESH_LIGHT_CLIENT_EVT_MAX (C++ enumerator), 451
- esp_ble_mesh_light_client_get_state (C++ function), 410
- ESP_BLE_MESH_LIGHT_CLIENT_GET_STATE_EVT (C++ enumerator), 451
- esp_ble_mesh_light_client_get_state_t (C++ union), 411
- esp_ble_mesh_light_client_get_state_t::lc_property (C++ member), 411
- ESP_BLE_MESH_LIGHT_CLIENT_PUBLISH_EVT (C++ enumerator), 451
- esp_ble_mesh_light_client_set_state (C++ function), 410
- ESP_BLE_MESH_LIGHT_CLIENT_SET_STATE_EVT (C++ enumerator), 451
- esp_ble_mesh_light_client_set_state_t (C++ union), 411
- esp_ble_mesh_light_client_set_state_t::ctl_default (C++ member), 411
- esp_ble_mesh_light_client_set_state_t::ctl_set (C++ member), 411
- esp_ble_mesh_light_client_set_state_t::ctl_temperature (C++ member), 411
- esp_ble_mesh_light_client_set_state_t::ctl_temperature (C++ member), 411
- esp_ble_mesh_light_client_set_state_t::hsl_default (C++ member), 412
- esp_ble_mesh_light_client_set_state_t::hsl_hue_set (C++ member), 411
- esp_ble_mesh_light_client_set_state_t::hsl_range (C++ member), 412
- esp_ble_mesh_light_client_set_state_t::hsl_saturation (C++ member), 412
- esp_ble_mesh_light_client_set_state_t::hsl_set (C++ member), 411
- esp_ble_mesh_light_client_set_state_t::lc_light_on (C++ member), 412
- esp_ble_mesh_light_client_set_state_t::lc_mode_set

(C++ member), 412
 esp_ble_mesh_light_client_set_state_t: esp_ble_mesh_light_client_status_cb_t::xyl_default
 (C++ member), 412 (C++ member), 413
 esp_ble_mesh_light_client_set_state_t: esp_ble_mesh_light_client_status_cb_t::xyl_range
 (C++ member), 412 (C++ member), 413
 esp_ble_mesh_light_client_set_state_t: esp_ble_mesh_light_client_status_cb_t::xyl_status
 (C++ member), 411 (C++ member), 413
 esp_ble_mesh_light_client_set_state_t: esp_ble_mesh_light_client_status_cb_t::xyl_target
 (C++ member), 411 (C++ member), 413
 esp_ble_mesh_light_client_set_state_t: ESP_BLE_MESH_LIGHT_CLIENT_TIMEOUT_EVT
 (C++ member), 411 (C++ enumerator), 451
 esp_ble_mesh_light_client_set_state_t: esp_ble_mesh_light_control_t (C++ class),
 (C++ member), 411 436
 esp_ble_mesh_light_client_set_state_t: esp_ble_mesh_light_control_t::prop_state
 (C++ member), 412 (C++ member), 437
 esp_ble_mesh_light_client_set_state_t: esp_ble_mesh_light_control_t::state
 (C++ member), 412 (C++ member), 437
 esp_ble_mesh_light_client_set_state_t: esp_ble_mesh_light_control_t::state_machine
 (C++ member), 412 (C++ member), 437
 esp_ble_mesh_light_client_status_cb_t esp_ble_mesh_light_ctl_default_set_t
 (C++ union), 412 (C++ class), 418
 esp_ble_mesh_light_client_status_cb_t: esp_ble_mesh_light_ctl_default_set_t::delta_uv
 (C++ member), 413 (C++ member), 418
 esp_ble_mesh_light_client_status_cb_t: esp_ble_mesh_light_ctl_default_set_t::lightness
 (C++ member), 412 (C++ member), 418
 esp_ble_mesh_light_client_status_cb_t: esp_ble_mesh_light_ctl_default_set_t::temperature
 (C++ member), 412 (C++ member), 418
 esp_ble_mesh_light_client_status_cb_t: esp_ble_mesh_light_ctl_default_status_cb_t
 (C++ member), 412 (C++ class), 423
 esp_ble_mesh_light_client_status_cb_t: esp_ble_mesh_light_ctl_default_status_cb_t::delta
 (C++ member), 413 (C++ member), 424
 esp_ble_mesh_light_client_status_cb_t: esp_ble_mesh_light_ctl_default_status_cb_t::light
 (C++ member), 413 (C++ member), 424
 esp_ble_mesh_light_client_status_cb_t: esp_ble_mesh_light_ctl_default_status_cb_t::tempe
 (C++ member), 413 (C++ member), 424
 esp_ble_mesh_light_client_status_cb_t: ESP_BLE_MESH_CTL_STATUS_LIGHTNESS_STATE
 (C++ member), 413 (C++ enumerator), 329
 esp_ble_mesh_light_client_status_cb_t: esp_ble_mesh_light_ctl_set_t (C++ class),
 (C++ member), 413 417
 esp_ble_mesh_light_client_status_cb_t: esp_ble_mesh_light_ctl_set_t::ctl_delta_uv
 (C++ member), 413 (C++ member), 417
 esp_ble_mesh_light_client_status_cb_t: esp_ble_mesh_light_ctl_set_t::ctl_lightness
 (C++ member), 413 (C++ member), 417
 esp_ble_mesh_light_client_status_cb_t: esp_ble_mesh_light_ctl_set_t::ctl_temperature
 (C++ member), 413 (C++ member), 417
 esp_ble_mesh_light_client_status_cb_t: esp_ble_mesh_light_ctl_set_t::delay
 (C++ member), 413 (C++ member), 417
 esp_ble_mesh_light_client_status_cb_t: esp_ble_mesh_light_ctl_set_t::op_en
 (C++ member), 413 (C++ member), 417
 esp_ble_mesh_light_client_status_cb_t: esp_ble_mesh_light_ctl_set_t::tid
 (C++ member), 412 (C++ member), 417
 esp_ble_mesh_light_client_status_cb_t: esp_ble_mesh_light_ctl_set_t::trans_time
 (C++ member), 412 (C++ member), 417
 esp_ble_mesh_light_client_status_cb_t: esp_ble_mesh_light_ctl_setup_srv_t
 (C++ member), 412 (C++ class), 430
 esp_ble_mesh_light_client_status_cb_t: esp_ble_mesh_light_ctl_setup_srv_t::model
 (C++ member), 412 (C++ member), 430
 esp_ble_mesh_light_client_status_cb_t: esp_ble_mesh_light_ctl_setup_srv_t::rsp_ctrl

(C++ member), 430
 esp_ble_mesh_light_ctl_setup_srv_t::state (C++ member), 430
 esp_ble_mesh_light_ctl_srv_t (C++ class), 429
 esp_ble_mesh_light_ctl_srv_t::last (C++ member), 430
 esp_ble_mesh_light_ctl_srv_t::model (C++ member), 430
 esp_ble_mesh_light_ctl_srv_t::rsp_ctrl (C++ member), 430
 esp_ble_mesh_light_ctl_srv_t::state (C++ member), 430
 esp_ble_mesh_light_ctl_srv_t::transition (C++ member), 430
 esp_ble_mesh_light_ctl_srv_t::tt_delta_delta_uv (C++ member), 430
 esp_ble_mesh_light_ctl_srv_t::tt_delta_delta_uv_mesh (C++ member), 430
 esp_ble_mesh_light_ctl_srv_t::tt_delta_delta_uv_temperature (C++ member), 430
 esp_ble_mesh_light_ctl_state_t (C++ class), 429
 esp_ble_mesh_light_ctl_state_t::delta_uv (C++ member), 429
 esp_ble_mesh_light_ctl_state_t::delta_uv_default (C++ member), 429
 esp_ble_mesh_light_ctl_state_t::lightness (C++ member), 429
 esp_ble_mesh_light_ctl_state_t::lightness_default (C++ member), 429
 esp_ble_mesh_light_ctl_state_t::status_code (C++ member), 429
 esp_ble_mesh_light_ctl_state_t::target_delta_uv (C++ member), 429
 esp_ble_mesh_light_ctl_state_t::target_delta_uv_mesh (C++ member), 429
 esp_ble_mesh_light_ctl_state_t::target_delta_uv_temperature (C++ member), 429
 esp_ble_mesh_light_ctl_state_t::temperature (C++ member), 429
 esp_ble_mesh_light_ctl_state_t::temperature_fault (C++ member), 429
 esp_ble_mesh_light_ctl_state_t::temperature_ranges_max (C++ member), 429
 esp_ble_mesh_light_ctl_state_t::temperature_ranges_min (C++ member), 429
 esp_ble_mesh_light_ctl_status_cb_t (C++ class), 422
 esp_ble_mesh_light_ctl_status_cb_t::op (C++ member), 423
 esp_ble_mesh_light_ctl_status_cb_t::presample (C++ member), 423
 esp_ble_mesh_light_ctl_status_cb_t::presample_temperature (C++ member), 423
 esp_ble_mesh_light_ctl_status_cb_t::resample_time (C++ member), 423
 esp_ble_mesh_light_ctl_status_cb_t::target_delta_uv (C++ member), 423
 esp_ble_mesh_light_ctl_status_cb_t::target_delta_uv_mesh (C++ member), 423
 esp_ble_mesh_light_ctl_status_cb_t::target_delta_uv_temperature (C++ member), 423
 esp_ble_mesh_light_ctl_status_cb_t::target_ctl_temperature (C++ member), 423
 ESP_BLE_MESH_LIGHT_CTL_TEMP_DELTA_UV_STATE (C++ enumerator), 329
 esp_ble_mesh_light_ctl_temp_srv_t (C++ class), 430
 esp_ble_mesh_light_ctl_temp_srv_t::last (C++ member), 430
 esp_ble_mesh_light_ctl_temp_srv_t::model (C++ member), 430
 esp_ble_mesh_light_ctl_temp_srv_t::rsp_ctrl (C++ member), 430
 esp_ble_mesh_light_ctl_temp_srv_t::state (C++ member), 430
 esp_ble_mesh_light_ctl_temperature_range_set_t (C++ class), 418
 esp_ble_mesh_light_ctl_temperature_range_set_t::r (C++ member), 418
 esp_ble_mesh_light_ctl_temperature_range_set_t::r_default (C++ member), 418
 esp_ble_mesh_light_ctl_temperature_range_status_cb_t (C++ class), 423
 esp_ble_mesh_light_ctl_temperature_range_status_cb_t::delta_uv (C++ member), 423
 esp_ble_mesh_light_ctl_temperature_range_status_cb_t::delta_uv_mesh (C++ member), 423
 esp_ble_mesh_light_ctl_temperature_range_status_cb_t::delta_uv_temperature (C++ member), 423
 esp_ble_mesh_light_ctl_temperature_set_t (C++ class), 417
 esp_ble_mesh_light_ctl_temperature_set_t::ctl_delta_uv (C++ member), 417
 esp_ble_mesh_light_ctl_temperature_set_t::ctl_delta_uv_mesh (C++ member), 417
 esp_ble_mesh_light_ctl_temperature_set_t::ctl_delta_uv_temperature (C++ member), 417
 esp_ble_mesh_light_ctl_temperature_set_t::delay (C++ member), 417
 esp_ble_mesh_light_ctl_temperature_set_t::op_en (C++ member), 417
 esp_ble_mesh_light_ctl_temperature_set_t::tid (C++ member), 417
 esp_ble_mesh_light_ctl_temperature_set_t::trans_t (C++ member), 417
 esp_ble_mesh_light_ctl_temperature_status_cb_t (C++ class), 423
 esp_ble_mesh_light_ctl_temperature_status_cb_t::op (C++ member), 423
 esp_ble_mesh_light_ctl_temperature_status_cb_t::presample (C++ member), 423
 esp_ble_mesh_light_ctl_temperature_status_cb_t::presample_temperature (C++ member), 423
 esp_ble_mesh_light_ctl_temperature_status_cb_t::resample_time (C++ member), 423
 esp_ble_mesh_light_ctl_temperature_status_cb_t::target_delta_uv (C++ member), 423
 esp_ble_mesh_light_ctl_temperature_status_cb_t::target_delta_uv_mesh (C++ member), 423
 esp_ble_mesh_light_ctl_temperature_status_cb_t::target_delta_uv_temperature (C++ member), 423
 esp_ble_mesh_light_ctl_temperature_status_cb_t::target_ctl_temperature (C++ member), 423

(C++ member), 436
 esp_ble_mesh_light_lc_state_machine_t::start_time (C++ member), 436
 esp_ble_mesh_light_lc_state_t (C++ class), 434
 esp_ble_mesh_light_lc_state_t::ambient_exp_level (C++ member), 434
 esp_ble_mesh_light_lc_state_t::light_offset (C++ member), 434
 esp_ble_mesh_light_lc_state_t::linear_output (C++ member), 434
 esp_ble_mesh_light_lc_state_t::mode (C++ member), 434
 esp_ble_mesh_light_lc_state_t::occupancy (C++ member), 434
 esp_ble_mesh_light_lc_state_t::occupancy_mode (C++ member), 434
 esp_ble_mesh_light_lc_state_t::target_light_level (C++ member), 434
 ESP_BLE_MESH_LIGHT_LIGHTNESS_ACTUAL_STATE (C++ enumerator), 329
 esp_ble_mesh_light_lightness_default_set_t (C++ class), 416
 esp_ble_mesh_light_lightness_default_set_t::delay (C++ member), 417
 esp_ble_mesh_light_lightness_default_set_t::model (C++ class), 422
 esp_ble_mesh_light_lightness_default_set_t::model_id (C++ member), 422
 esp_ble_mesh_light_lightness_last_status_report (C++ class), 422
 esp_ble_mesh_light_lightness_last_status_report::mesh_light_lightness_setup_srv_t::state (C++ member), 422
 esp_ble_mesh_light_lightness_linear_set_t (C++ class), 416
 esp_ble_mesh_light_lightness_linear_set_t::delay (C++ member), 416
 esp_ble_mesh_light_lightness_linear_set_t::brightness (C++ member), 416
 esp_ble_mesh_light_lightness_linear_set_t::mode (C++ member), 416
 esp_ble_mesh_light_lightness_linear_set_t::model (C++ member), 416
 esp_ble_mesh_light_lightness_linear_set_t::transition_time (C++ member), 416
 ESP_BLE_MESH_LIGHT_LIGHTNESS_LINEAR_STATE (C++ enumerator), 329
 esp_ble_mesh_light_lightness_linear_status_report (C++ class), 422
 esp_ble_mesh_light_lightness_linear_status_report::mesh_light_lightness_srv_t::tt_delta_lightness (C++ member), 422
 esp_ble_mesh_light_lightness_linear_status_report::mesh_light_lightness_srv_t::tt_delta_lightness (C++ member), 422
 esp_ble_mesh_light_lightness_linear_status_report::mesh_light_lightness_state_t (C++ member), 422
 esp_ble_mesh_light_lightness_linear_status_report::mesh_light_lightness_state_t::lightness_actual (C++ member), 422
 esp_ble_mesh_light_lightness_linear_status_report::mesh_light_lightness_state_t::lightness_delta (C++ member), 422
 esp_ble_mesh_light_lightness_range_set_t (C++ class), 417
 esp_ble_mesh_light_lightness_range_set_t::range_min (C++ member), 417
 esp_ble_mesh_light_lightness_range_set_t::range_max (C++ member), 417
 esp_ble_mesh_light_lightness_range_status_cb_t (C++ class), 422
 esp_ble_mesh_light_lightness_range_status_cb_t::range_min (C++ member), 422
 esp_ble_mesh_light_lightness_range_status_cb_t::range_max (C++ member), 422
 esp_ble_mesh_light_lightness_set_t (C++ class), 416
 esp_ble_mesh_light_lightness_set_t::delay (C++ member), 416
 esp_ble_mesh_light_lightness_set_t::lightness (C++ member), 416
 esp_ble_mesh_light_lightness_set_t::open (C++ member), 416
 esp_ble_mesh_light_lightness_set_t::tid (C++ member), 416
 esp_ble_mesh_light_lightness_set_t::trans_time (C++ member), 416
 esp_ble_mesh_light_lightness_setup_srv_t (C++ class), 429
 esp_ble_mesh_light_lightness_setup_srv_t::model (C++ member), 429
 esp_ble_mesh_light_lightness_setup_srv_t::rsp_ctrl (C++ member), 429
 esp_ble_mesh_light_lightness_setup_srv_t::state (C++ member), 429
 esp_ble_mesh_light_lightness_srv_t (C++ class), 428
 esp_ble_mesh_light_lightness_srv_t::actual_transition_time (C++ member), 428
 esp_ble_mesh_light_lightness_srv_t::last (C++ member), 428
 esp_ble_mesh_light_lightness_srv_t::linear_transition_time (C++ member), 428
 esp_ble_mesh_light_lightness_srv_t::model (C++ member), 428
 esp_ble_mesh_light_lightness_srv_t::rsp_ctrl (C++ member), 428
 esp_ble_mesh_light_lightness_srv_t::state (C++ member), 428
 esp_ble_mesh_light_lightness_srv_t::tt_delta_lightness (C++ member), 428
 esp_ble_mesh_light_lightness_state_t (C++ class), 428
 esp_ble_mesh_light_lightness_state_t::lightness_actual (C++ member), 428
 esp_ble_mesh_light_lightness_state_t::lightness_delta (C++ member), 428
 esp_ble_mesh_light_lightness_state_t::lightness_linear (C++ member), 428

- (C++ member), 428
- esp_ble_mesh_light_lightness_state_t::lightness [esp_ble_mesh_light_xyl_range_status_cb_t::xyl_x](#)
(C++ member), 428 (C++ member), 426
- esp_ble_mesh_light_lightness_state_t::lightness [esp_ble_mesh_light_xyl_range_status_cb_t::xyl_y](#)
(C++ member), 428 (C++ member), 427
- esp_ble_mesh_light_lightness_state_t::lightness [esp_ble_mesh_light_xyl_range_status_cb_t::xyl_z](#)
(C++ member), 428 (C++ member), 427
- esp_ble_mesh_light_lightness_state_t::status_code [esp_ble_mesh_light_xyl_set_t](#) (C++ class),
(C++ member), 428 419
- esp_ble_mesh_light_lightness_state_t::transition [esp_ble_mesh_light_xyl_set_t::delay](#)
(C++ member), 428 (C++ member), 420
- esp_ble_mesh_light_lightness_state_t::transition [esp_ble_mesh_light_xyl_set_t::op_en](#)
(C++ member), 428 (C++ member), 420
- esp_ble_mesh_light_lightness_status_cb [esp_ble_mesh_light_xyl_set_t::tid](#)
(C++ class), 421 (C++ member), 420
- esp_ble_mesh_light_lightness_status_cb [esp_ble_mesh_light_xyl_set_t::trans_time](#)
(C++ member), 422 (C++ member), 420
- esp_ble_mesh_light_lightness_status_cb [esp_ble_mesh_light_xyl_set_t::xyl_lightness](#)
(C++ member), 422 (C++ member), 420
- esp_ble_mesh_light_lightness_status_cb [esp_ble_mesh_light_xyl_set_t::xyl_x](#)
(C++ member), 422 (C++ member), 420
- esp_ble_mesh_light_lightness_status_cb [esp_ble_mesh_light_xyl_set_t::xyl_y](#)
(C++ member), 422 (C++ member), 420
- esp_ble_mesh_light_message_opcode_t [esp_ble_mesh_light_xyl_setup_srv_t](#)
(C++ type), 324 (C++ class), 434
- esp_ble_mesh_light_xyl_default_set_t [esp_ble_mesh_light_xyl_setup_srv_t::model](#)
(C++ class), 420 (C++ member), 434
- esp_ble_mesh_light_xyl_default_set_t::lightness [esp_ble_mesh_light_xyl_setup_srv_t::rsp_ctrl](#)
(C++ member), 420 (C++ member), 434
- esp_ble_mesh_light_xyl_default_set_t::xyl_x [esp_ble_mesh_light_xyl_setup_srv_t::state](#)
(C++ member), 420 (C++ member), 434
- esp_ble_mesh_light_xyl_default_set_t::xyl_y [esp_ble_mesh_light_xyl_srv_t](#) (C++ class),
(C++ member), 420 433
- esp_ble_mesh_light_xyl_default_status_cb [esp_ble_mesh_light_xyl_srv_t::last](#)
(C++ class), 426 (C++ member), 434
- esp_ble_mesh_light_xyl_default_status_cb [esp_ble_mesh_light_xyl_srv_t::model](#)
(C++ member), 426 (C++ member), 433
- esp_ble_mesh_light_xyl_default_status_cb [esp_ble_mesh_light_xyl_srv_t::rsp_ctrl](#)
(C++ member), 426 (C++ member), 433
- esp_ble_mesh_light_xyl_default_status_cb [esp_ble_mesh_light_xyl_srv_t::state](#)
(C++ member), 426 (C++ member), 433
- ESP_BLE_MESH_LIGHT_XYL_LIGHTNESS_STATE [esp_ble_mesh_light_xyl_srv_t::transition](#)
(C++ enumerator), 330 (C++ member), 434
- esp_ble_mesh_light_xyl_range_set_t [esp_ble_mesh_light_xyl_srv_t::tt_delta_lightness](#)
(C++ class), 420 (C++ member), 434
- esp_ble_mesh_light_xyl_range_set_t::xyl_x [esp_ble_mesh_light_xyl_srv_t::tt_delta_x](#)
(C++ member), 420 (C++ member), 434
- esp_ble_mesh_light_xyl_range_set_t::xyl_y [esp_ble_mesh_light_xyl_srv_t::tt_delta_y](#)
(C++ member), 420 (C++ member), 434
- esp_ble_mesh_light_xyl_range_set_t::xyl_z [esp_ble_mesh_light_xyl_state_t](#) (C++
(C++ member), 420 class), 433
- esp_ble_mesh_light_xyl_range_set_t::lightness [esp_ble_mesh_light_xyl_state_t::lightness](#)
(C++ member), 420 (C++ member), 433
- esp_ble_mesh_light_xyl_range_status_cb [esp_ble_mesh_light_xyl_state_t::lightness_default](#)
(C++ class), 426 (C++ member), 433
- esp_ble_mesh_light_xyl_range_status_cb [esp_ble_mesh_light_xyl_state_t::status_code](#)
(C++ member), 426 (C++ member), 433
- esp_ble_mesh_light_xyl_range_status_cb [esp_ble_mesh_light_xyl_state_t::target_lightness](#)

- esp_ble_mesh_model_cb_param_t::ble_mesh_model_receive_subtidal_model_param_t::model_server_model_update
 (C++ member), 299 (C++ member), 298
- esp_ble_mesh_model_cb_param_t::ble_mesh_model_receive_subtidal_model_top(C++ type), 341
 (C++ member), 299 esp_ble_mesh_model_cbs_t (C++ class), 302
- esp_ble_mesh_model_cb_param_t::ble_mesh_model_receive_subtidal_model_param_t::topcode
 (C++ member), 299 (C++ member), 302
- esp_ble_mesh_model_cb_param_t::ble_mesh_model_receive_subtidal_model_param_t::topcode
 (C++ class), 299 ESP_BLE_MESH_MODEL_CFG_GEN (C macro), 370
 (C++ member), 299 ESP_BLE_MESH_MODEL_CFG_SRV (C macro), 370
- esp_ble_mesh_model_cb_param_t::ble_mesh_model_receive_subtidal_model_param_t::topcode
 (C++ member), 299 ESP_BLE_MESH_MODEL_EVT_PARAM (C++ enumerator), 330
- esp_ble_mesh_model_cb_param_t::ble_mesh_model_receive_subtidal_model_param_t::topcode
 (C++ member), 299 ESP_BLE_MESH_MODEL_GEN_ADMIN_PROP_SRV (C macro), 400
- esp_ble_mesh_model_cb_param_t::ble_mesh_model_receive_subtidal_model_param_t::topcode
 (C++ member), 299 ESP_BLE_MESH_MODEL_GEN_BATTERY_CELL (C macro), 397
- esp_ble_mesh_model_cb_param_t::ble_mesh_model_receive_subtidal_model_param_t::topcode
 (C++ member), 299 ESP_BLE_MESH_MODEL_GEN_BATTERY_SRV (C macro), 399
- esp_ble_mesh_model_cb_param_t::ble_mesh_model_receive_subtidal_model_param_t::topcode
 (C++ member), 299 ESP_BLE_MESH_MODEL_GEN_CHARACTERISTIC_PROP_SRV (C macro), 400
- esp_ble_mesh_model_cb_param_t::ble_mesh_model_receive_subtidal_model_param_t::topcode
 (C++ class), 299 ESP_BLE_MESH_MODEL_GEN_DEF_TRANS_TIME_CLI (C macro), 397
- esp_ble_mesh_model_cb_param_t::ble_mesh_model_receive_subtidal_model_param_t::topcode
 (C++ member), 299 ESP_BLE_MESH_MODEL_GEN_DEF_TRANS_TIME_SRV (C macro), 398
- esp_ble_mesh_model_cb_param_t::ble_mesh_model_receive_subtidal_model_param_t::topcode
 (C++ member), 299 ESP_BLE_MESH_MODEL_GEN_DEF_UUID_CELL (C macro), 397
- esp_ble_mesh_model_cb_param_t::ble_mesh_model_receive_subtidal_model_param_t::topcode
 (C++ class), 299 ESP_BLE_MESH_MODEL_GEN_EV_EVT_SRV (C macro), 398
- esp_ble_mesh_model_cb_param_t::ble_mesh_model_receive_subtidal_model_param_t::topcode
 (C++ member), 300 ESP_BLE_MESH_MODEL_GEN_EV_OCCUPANCY_CELL (C macro), 397
- esp_ble_mesh_model_cb_param_t::ble_mesh_model_receive_subtidal_model_param_t::topcode
 (C++ class), 300 ESP_BLE_MESH_MODEL_GEN_LOCATION_SETUP_SRV (C macro), 399
- esp_ble_mesh_model_cb_param_t::ble_mesh_model_receive_subtidal_model_param_t::topcode
 (C++ member), 300 ESP_BLE_MESH_MODEL_GEN_LOCATION_SRV (C macro), 399
- esp_ble_mesh_model_cb_param_t::ble_mesh_model_receive_subtidal_model_param_t::topcode
 (C++ member), 300 ESP_BLE_MESH_MODEL_GEN_MANUFACTURER_PROP_SRV (C macro), 400
- esp_ble_mesh_model_cb_param_t::ble_mesh_model_receive_subtidal_model_param_t::topcode
 (C++ member), 300 ESP_BLE_MESH_MODEL_GEN_MODEL_CLI (C macro), 397
- esp_ble_mesh_model_cb_param_t::ble_mesh_model_receive_subtidal_model_param_t::topcode
 (C++ member), 300 ESP_BLE_MESH_MODEL_GEN_MODEL_SRV (C macro), 398
- esp_ble_mesh_model_cb_param_t::ble_mesh_model_receive_subtidal_model_param_t::topcode
 (C++ class), 300 ESP_BLE_MESH_MODEL_GEN_NETWORK_POWER_COMP_EVT_CELL (C macro), 397
- esp_ble_mesh_model_cb_param_t::ble_mesh_model_receive_subtidal_model_param_t::topcode
 (C++ member), 300 ESP_BLE_MESH_MODEL_GEN_NETWORK_POWER_COMP_EVT_CELL_SRV_CODE (C macro), 399
- esp_ble_mesh_model_cb_param_t::ble_mesh_model_receive_subtidal_model_param_t::topcode
 (C++ member), 300 ESP_BLE_MESH_MODEL_GEN_NETWORK_POWER_COMP_EVT_CELL_SRV_CODE : model (C macro), 399
- esp_ble_mesh_model_cb_param_t::ble_mesh_model_receive_subtidal_model_param_t::topcode
 (C++ member), 300 ESP_BLE_MESH_MODEL_GEN_NETWORK_POWER_COMP_EVT_CELL_SRV_CODE : type (C macro), 397
- esp_ble_mesh_model_cb_param_t::client_srv_ble_mesh_model_gen_power_onoff_setup_srv
 (C++ member), 298 (C macro), 398
- esp_ble_mesh_model_cb_param_t::client_srv_ble_mesh_model_gen_power_onoff_srv
 (C++ member), 298 (C macro), 398
- esp_ble_mesh_model_cb_param_t::model_ops_ble_mesh_model_gen_property_cli (C macro), 398
 (C++ member), 298 (C++ member), 398
- esp_ble_mesh_model_cb_param_t::model_ops_ble_mesh_model_gen_user_prop_srv
 (C++ member), 298 (C macro), 400
- esp_ble_mesh_model_cb_param_t::model_ops_ble_mesh_model_health_cli (C macro),
 (C++ member), 298 408
- esp_ble_mesh_model_cb_param_t::model_ops_ble_mesh_model_health_srv (C macro),
 (C++ member), 298 407

| | |
|--|--|
| ESP_BLE_MESH_MODEL_ID_CONFIG_CLI (C macro), 312 | ESP_BLE_MESH_MODEL_ID_LIGHT_CTL_TEMP_SRV (C macro), 313 |
| ESP_BLE_MESH_MODEL_ID_CONFIG_SRV (C macro), 312 | ESP_BLE_MESH_MODEL_ID_LIGHT_HSL_CLI (C macro), 313 |
| ESP_BLE_MESH_MODEL_ID_GEN_ADMIN_PROP_SRV (C macro), 312 | ESP_BLE_MESH_MODEL_ID_LIGHT_HSL_HUE_SRV (C macro), 313 |
| ESP_BLE_MESH_MODEL_ID_GEN_BATTERY_CLI (C macro), 312 | ESP_BLE_MESH_MODEL_ID_LIGHT_HSL_SAT_SRV (C macro), 313 |
| ESP_BLE_MESH_MODEL_ID_GEN_BATTERY_SRV (C macro), 312 | ESP_BLE_MESH_MODEL_ID_LIGHT_HSL_SETUP_SRV (C macro), 313 |
| ESP_BLE_MESH_MODEL_ID_GEN_CLIENT_PROP_SRV (C macro), 312 | ESP_BLE_MESH_MODEL_ID_LIGHT_HSL_SRV (C macro), 313 |
| ESP_BLE_MESH_MODEL_ID_GEN_DEF_TRANS_TIME_CLI (C macro), 312 | ESP_BLE_MESH_MODEL_ID_LIGHT_LC_CLI (C macro), 313 |
| ESP_BLE_MESH_MODEL_ID_GEN_DEF_TRANS_TIME_SRV (C macro), 312 | ESP_BLE_MESH_MODEL_ID_LIGHT_LC_SETUP_SRV (C macro), 313 |
| ESP_BLE_MESH_MODEL_ID_GEN_LEVEL_CLI (C macro), 312 | ESP_BLE_MESH_MODEL_ID_LIGHT_LC_SRV (C macro), 313 |
| ESP_BLE_MESH_MODEL_ID_GEN_LEVEL_SRV (C macro), 312 | ESP_BLE_MESH_MODEL_ID_LIGHT_LIGHTNESS_CLI (C macro), 313 |
| ESP_BLE_MESH_MODEL_ID_GEN_LOCATION_CLI (C macro), 312 | ESP_BLE_MESH_MODEL_ID_LIGHT_LIGHTNESS_SETUP_SRV (C macro), 313 |
| ESP_BLE_MESH_MODEL_ID_GEN_LOCATION_SETUP_SRV (C macro), 312 | ESP_BLE_MESH_MODEL_ID_LIGHT_LIGHTNESS_SRV (C macro), 313 |
| ESP_BLE_MESH_MODEL_ID_GEN_LOCATION_SRV (C macro), 312 | ESP_BLE_MESH_MODEL_ID_LIGHT_XYL_CLI (C macro), 313 |
| ESP_BLE_MESH_MODEL_ID_GEN_MANUFACTURER_PROP_SRV (C macro), 312 | ESP_BLE_MESH_MODEL_ID_LIGHT_XYL_SETUP_SRV (C macro), 313 |
| ESP_BLE_MESH_MODEL_ID_GEN_ONOFF_CLI (C macro), 312 | ESP_BLE_MESH_MODEL_ID_LIGHT_XYL_SRV (C macro), 313 |
| ESP_BLE_MESH_MODEL_ID_GEN_ONOFF_SRV (C macro), 312 | ESP_BLE_MESH_MODEL_ID_SCENE_CLI (C macro), 313 |
| ESP_BLE_MESH_MODEL_ID_GEN_POWER_LEVEL_CLI (C macro), 312 | ESP_BLE_MESH_MODEL_ID_SCENE_SETUP_SRV (C macro), 312 |
| ESP_BLE_MESH_MODEL_ID_GEN_POWER_LEVEL_SETUP_SRV (C macro), 312 | ESP_BLE_MESH_MODEL_ID_SCENE_SRV (C macro), 312 |
| ESP_BLE_MESH_MODEL_ID_GEN_POWER_LEVEL_SRV (C macro), 312 | ESP_BLE_MESH_MODEL_ID_SCHEDULER_CLI (C macro), 313 |
| ESP_BLE_MESH_MODEL_ID_GEN_POWER_ONOFF_CLI (C macro), 312 | ESP_BLE_MESH_MODEL_ID_SCHEDULER_SETUP_SRV (C macro), 313 |
| ESP_BLE_MESH_MODEL_ID_GEN_POWER_ONOFF_SETUP_SRV (C macro), 312 | ESP_BLE_MESH_MODEL_ID_SCHEDULER_SRV (C macro), 313 |
| ESP_BLE_MESH_MODEL_ID_GEN_POWER_ONOFF_SRV (C macro), 312 | ESP_BLE_MESH_MODEL_ID_SENSOR_CLI (C macro), 312 |
| ESP_BLE_MESH_MODEL_ID_GEN_PROP_CLI (C macro), 312 | ESP_BLE_MESH_MODEL_ID_SENSOR_SETUP_SRV (C macro), 312 |
| ESP_BLE_MESH_MODEL_ID_GEN_USER_PROP_SRV (C macro), 312 | ESP_BLE_MESH_MODEL_ID_SENSOR_SRV (C macro), 312 |
| ESP_BLE_MESH_MODEL_ID_HEALTH_CLI (C macro), 312 | ESP_BLE_MESH_MODEL_ID_TIME_CLI (C macro), 312 |
| ESP_BLE_MESH_MODEL_ID_HEALTH_SRV (C macro), 312 | ESP_BLE_MESH_MODEL_ID_TIME_SETUP_SRV (C macro), 312 |
| ESP_BLE_MESH_MODEL_ID_LIGHT_CTL_CLI (C macro), 313 | ESP_BLE_MESH_MODEL_ID_TIME_SRV (C macro), 312 |
| ESP_BLE_MESH_MODEL_ID_LIGHT_CTL_SETUP_SRV (C macro), 313 | ESP_BLE_MESH_MODEL_LIGHT_CTL_CLI (C macro), 448 |
| ESP_BLE_MESH_MODEL_ID_LIGHT_CTL_SRV (C macro), 313 | ESP_BLE_MESH_MODEL_LIGHT_CTL_SETUP_SRV (C macro), 449 |

- ESP_BLE_MESH_MODEL_LIGHT_CTL_SRV (C macro), 448
- ESP_BLE_MESH_MODEL_LIGHT_CTL_TEMP_SRV (C macro), 449
- ESP_BLE_MESH_MODEL_LIGHT_HSL_CLI (C macro), 448
- ESP_BLE_MESH_MODEL_LIGHT_HSL_HUE_SRV (C macro), 450
- ESP_BLE_MESH_MODEL_LIGHT_HSL_SAT_SRV (C macro), 450
- ESP_BLE_MESH_MODEL_LIGHT_HSL_SETUP_SRV (C macro), 449
- ESP_BLE_MESH_MODEL_LIGHT_HSL_SRV (C macro), 449
- ESP_BLE_MESH_MODEL_LIGHT_LC_CLI (C macro), 448
- ESP_BLE_MESH_MODEL_LIGHT_LC_SETUP_SRV (C macro), 450
- ESP_BLE_MESH_MODEL_LIGHT_LC_SRV (C macro), 450
- ESP_BLE_MESH_MODEL_LIGHT_LIGHTNESS_CLI (C macro), 447
- ESP_BLE_MESH_MODEL_LIGHT_LIGHTNESS_SETUP_SRV (C macro), 448
- ESP_BLE_MESH_MODEL_LIGHT_LIGHTNESS_SRV (C macro), 448
- ESP_BLE_MESH_MODEL_LIGHT_XYL_CLI (C macro), 448
- ESP_BLE_MESH_MODEL_LIGHT_XYL_SETUP_SRV (C macro), 450
- ESP_BLE_MESH_MODEL_LIGHT_XYL_SRV (C macro), 450
- esp_ble_mesh_model_msg_opcode_init (C++ function), 334
- ESP_BLE_MESH_MODEL_NONE (C macro), 311
- ESP_BLE_MESH_MODEL_OP (C macro), 311
- ESP_BLE_MESH_MODEL_OP_1 (C macro), 310
- ESP_BLE_MESH_MODEL_OP_2 (C macro), 310
- ESP_BLE_MESH_MODEL_OP_3 (C macro), 310
- ESP_BLE_MESH_MODEL_OP_APP_KEY_ADD (C macro), 314
- ESP_BLE_MESH_MODEL_OP_APP_KEY_DELETE (C macro), 315
- ESP_BLE_MESH_MODEL_OP_APP_KEY_GET (C macro), 314
- ESP_BLE_MESH_MODEL_OP_APP_KEY_LIST (C macro), 315
- ESP_BLE_MESH_MODEL_OP_APP_KEY_STATUS (C macro), 315
- ESP_BLE_MESH_MODEL_OP_APP_KEY_UPDATE (C macro), 315
- ESP_BLE_MESH_MODEL_OP_ATTENTION_GET (C macro), 316
- ESP_BLE_MESH_MODEL_OP_ATTENTION_SET (C macro), 317
- ESP_BLE_MESH_MODEL_OP_ATTENTION_SET_UNACK (C macro), 317
- ESP_BLE_MESH_MODEL_OP_ATTENTION_STATUS (C macro), 317
- (C macro), 317
- ESP_BLE_MESH_MODEL_OP_BEACON_GET (C macro), 313
- ESP_BLE_MESH_MODEL_OP_BEACON_SET (C macro), 314
- ESP_BLE_MESH_MODEL_OP_BEACON_STATUS (C macro), 315
- ESP_BLE_MESH_MODEL_OP_COMPOSITION_DATA_GET (C macro), 313
- ESP_BLE_MESH_MODEL_OP_COMPOSITION_DATA_STATUS (C macro), 315
- ESP_BLE_MESH_MODEL_OP_DEFAULT_TTL_GET (C macro), 313
- ESP_BLE_MESH_MODEL_OP_DEFAULT_TTL_SET (C macro), 314
- ESP_BLE_MESH_MODEL_OP_DEFAULT_TTL_STATUS (C macro), 315
- ESP_BLE_MESH_MODEL_OP_END (C macro), 311
- ESP_BLE_MESH_MODEL_OP_FRIEND_GET (C macro), 313
- ESP_BLE_MESH_MODEL_OP_FRIEND_SET (C macro), 315
- ESP_BLE_MESH_MODEL_OP_FRIEND_STATUS (C macro), 316
- ESP_BLE_MESH_MODEL_OP_GATT_PROXY_GET (C macro), 313
- ESP_BLE_MESH_MODEL_OP_GATT_PROXY_SET (C macro), 314
- ESP_BLE_MESH_MODEL_OP_GATT_PROXY_STATUS (C macro), 315
- ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTIES_GET (C macro), 318
- ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTIES_STATUS (C macro), 318
- ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTY_GET (C macro), 318
- ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTY_SET (C macro), 318
- ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTY_SET_UNACK (C macro), 318
- ESP_BLE_MESH_MODEL_OP_GEN_ADMIN_PROPERTY_STATUS (C macro), 318
- ESP_BLE_MESH_MODEL_OP_GEN_BATTERY_GET (C macro), 318
- ESP_BLE_MESH_MODEL_OP_GEN_BATTERY_STATUS (C macro), 318
- ESP_BLE_MESH_MODEL_OP_GEN_CLIENT_PROPERTIES_GET (C macro), 319
- ESP_BLE_MESH_MODEL_OP_GEN_CLIENT_PROPERTIES_STATUS (C macro), 319
- ESP_BLE_MESH_MODEL_OP_GEN_DEF_TRANS_TIME_GET (C macro), 317
- ESP_BLE_MESH_MODEL_OP_GEN_DEF_TRANS_TIME_SET (C macro), 317
- ESP_BLE_MESH_MODEL_OP_GEN_DEF_TRANS_TIME_SET_UNACK (C macro), 317
- ESP_BLE_MESH_MODEL_OP_GEN_DEF_TRANS_TIME_STATUS (C macro), 317

ESP_BLE_MESH_MODEL_OP_GEN_DELTA_SET (C ESP_BLE_MESH_MODEL_OP_GEN_ONPOWERUP_STATUS
 macro), 317 (C macro), 317
 ESP_BLE_MESH_MODEL_OP_GEN_DELTA_SET_UNACK ESP_BLE_MESH_MODEL_OP_GEN_POWER_DEFAULT_GET
 (C macro), 317 (C macro), 318
 ESP_BLE_MESH_MODEL_OP_GEN_LEVEL_GET (C ESP_BLE_MESH_MODEL_OP_GEN_POWER_DEFAULT_SET
 macro), 317 (C macro), 318
 ESP_BLE_MESH_MODEL_OP_GEN_LEVEL_SET (C ESP_BLE_MESH_MODEL_OP_GEN_POWER_DEFAULT_SET_UNACK
 macro), 317 (C macro), 318
 ESP_BLE_MESH_MODEL_OP_GEN_LEVEL_SET_UNACK ESP_BLE_MESH_MODEL_OP_GEN_POWER_DEFAULT_STATUS
 (C macro), 317 (C macro), 318
 ESP_BLE_MESH_MODEL_OP_GEN_LEVEL_STATUS ESP_BLE_MESH_MODEL_OP_GEN_POWER_LAST_GET
 (C macro), 317 (C macro), 318
 ESP_BLE_MESH_MODEL_OP_GEN_LOC_GLOBAL_GET ESP_BLE_MESH_MODEL_OP_GEN_POWER_LAST_STATUS
 (C macro), 318 (C macro), 318
 ESP_BLE_MESH_MODEL_OP_GEN_LOC_GLOBAL_SET ESP_BLE_MESH_MODEL_OP_GEN_POWER_LEVEL_GET
 (C macro), 318 (C macro), 317
 ESP_BLE_MESH_MODEL_OP_GEN_LOC_GLOBAL_SET_UNACK ESP_BLE_MESH_MODEL_OP_GEN_POWER_LEVEL_SET
 (C macro), 318 (C macro), 317
 ESP_BLE_MESH_MODEL_OP_GEN_LOC_GLOBAL_STATUS ESP_BLE_MESH_MODEL_OP_GEN_POWER_LEVEL_SET_UNACK
 (C macro), 318 (C macro), 317
 ESP_BLE_MESH_MODEL_OP_GEN_LOC_LOCAL_GET ESP_BLE_MESH_MODEL_OP_GEN_POWER_LEVEL_STATUS
 (C macro), 318 (C macro), 317
 ESP_BLE_MESH_MODEL_OP_GEN_LOC_LOCAL_SET ESP_BLE_MESH_MODEL_OP_GEN_POWER_RANGE_GET
 (C macro), 318 (C macro), 318
 ESP_BLE_MESH_MODEL_OP_GEN_LOC_LOCAL_SET_UNACK ESP_BLE_MESH_MODEL_OP_GEN_POWER_RANGE_SET
 (C macro), 318 (C macro), 318
 ESP_BLE_MESH_MODEL_OP_GEN_LOC_LOCAL_STATUS ESP_BLE_MESH_MODEL_OP_GEN_POWER_RANGE_SET_UNACK
 (C macro), 318 (C macro), 318
 ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTY_MESH_GET ESP_BLE_MESH_MODEL_OP_GEN_POWER_RANGE_STATUS
 (C macro), 318 (C macro), 318
 ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTY_MESH_STATUS ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTIES_GET
 (C macro), 318 (C macro), 318
 ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTY_MESH_UNACK ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTIES_STATUS
 (C macro), 318 (C macro), 318
 ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTY_MESH_GET ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTY_GET
 (C macro), 318 (C macro), 319
 ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTY_MESH_UNACK ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTY_SET
 (C macro), 318 (C macro), 319
 ESP_BLE_MESH_MODEL_OP_GEN_MANUFACTURER_PROPERTY_MESH_STATUS ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTY_SET_UNACK
 (C macro), 318 (C macro), 319
 ESP_BLE_MESH_MODEL_OP_GEN_MOVE_SET (C ESP_BLE_MESH_MODEL_OP_GEN_USER_PROPERTY_STATUS
 macro), 317 (C macro), 319
 ESP_BLE_MESH_MODEL_OP_GEN_MOVE_SET_UNACK ESP_BLE_MESH_MODEL_OP_HEALTH_CURRENT_STATUS
 (C macro), 317 (C macro), 317
 ESP_BLE_MESH_MODEL_OP_GEN_ONOFF_GET (C ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_CLEAR
 macro), 317 (C macro), 316
 ESP_BLE_MESH_MODEL_OP_GEN_ONOFF_SET (C ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_CLEAR_UNACK
 macro), 317 (C macro), 316
 ESP_BLE_MESH_MODEL_OP_GEN_ONOFF_SET_UNACK ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_GET
 (C macro), 317 (C macro), 316
 ESP_BLE_MESH_MODEL_OP_GEN_ONOFF_STATUS ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_STATUS
 (C macro), 317 (C macro), 317
 ESP_BLE_MESH_MODEL_OP_GEN_ONPOWERUP_GET ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_TEST
 (C macro), 317 (C macro), 316
 ESP_BLE_MESH_MODEL_OP_GEN_ONPOWERUP_SET ESP_BLE_MESH_MODEL_OP_HEALTH_FAULT_TEST_UNACK
 (C macro), 317 (C macro), 316
 ESP_BLE_MESH_MODEL_OP_GEN_ONPOWERUP_SET_UNACK ESP_BLE_MESH_MODEL_OP_HEALTH_PERIOD_GET
 (C macro), 317 (C macro), 316

ESP_BLE_MESH_MODEL_OP_LIGHT_LC_OM_GET (C macro), 322
 ESP_BLE_MESH_MODEL_OP_LIGHT_LC_OM_SET (C macro), 322
 ESP_BLE_MESH_MODEL_OP_LIGHT_LC_OM_SET_UNACK (C macro), 322
 ESP_BLE_MESH_MODEL_OP_LIGHT_LC_OM_STATUS (C macro), 322
 ESP_BLE_MESH_MODEL_OP_LIGHT_LC_PROPERTY_GET (C macro), 322
 ESP_BLE_MESH_MODEL_OP_LIGHT_LC_PROPERTY_SET (C macro), 322
 ESP_BLE_MESH_MODEL_OP_LIGHT_LC_PROPERTY_SET_UNACK (C macro), 322
 ESP_BLE_MESH_MODEL_OP_LIGHT_LC_PROPERTY_STATUS (C macro), 322
 ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_DEFAULT_GET (C macro), 320
 ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_DEFAULT_SET (C macro), 320
 ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_DEFAULT_STATUS (C macro), 320
 ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_ENABLE (C macro), 320
 ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_DISABLE (C macro), 320
 ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_STATUS (C macro), 320
 ESP_BLE_MESH_MODEL_OP_LIGHT_LIGHTNESS_UNACK (C macro), 320
 ESP_BLE_MESH_MODEL_OP_LP_NACK (C macro), 314
 ESP_BLE_MESH_MODEL_OP_LP_POLLTIMEOUT_GET (C macro), 316
 ESP_BLE_MESH_MODEL_OP_LP_POLLTIMEOUT_STATUS (C macro), 316
 ESP_BLE_MESH_MODEL_OP_MODEL_APP_BIND (C macro), 314
 ESP_BLE_MESH_MODEL_OP_MODEL_APP_STATUS (C macro), 315
 ESP_BLE_MESH_MODEL_OP_MODEL_APP_UNBIND (C macro), 315
 ESP_BLE_MESH_MODEL_OP_MODEL_PUB_GET (C macro), 313
 ESP_BLE_MESH_MODEL_OP_MODEL_PUB_SET (C macro), 314
 ESP_BLE_MESH_MODEL_OP_MODEL_PUB_STATUS (C macro), 315
 ESP_BLE_MESH_MODEL_OP_MODEL_PUB_VIRTUAL_ADDR_SET (C macro), 315
 ESP_BLE_MESH_MODEL_OP_MODEL_SUB_ADD (C macro), 314
 ESP_BLE_MESH_MODEL_OP_MODEL_SUB_DELETE (C macro), 314
 ESP_BLE_MESH_MODEL_OP_MODEL_SUB_DELETE_ALL (C macro), 315
 ESP_BLE_MESH_MODEL_OP_MODEL_SUB_OVERWRITE (C macro), 314
 ESP_BLE_MESH_MODEL_OP_MODEL_SUB_STATUS (C macro), 315
 ESP_BLE_MESH_MODEL_OP_MODEL_SUB_VIRTUAL_ADDR_ADD (C macro), 314
 ESP_BLE_MESH_MODEL_OP_MODEL_SUB_VIRTUAL_ADDR_DELETE (C macro), 314
 ESP_BLE_MESH_MODEL_OP_MODEL_SUB_VIRTUAL_ADDR_OVERWRITE (C macro), 314
 ESP_BLE_MESH_MODEL_OP_NET_KEY_ADD (C macro), 314

ESP_BLE_MESH_MODEL_OP_NET_KEY_DELETE (C macro), 315
 ESP_BLE_MESH_MODEL_OP_NET_KEY_GET (C macro), 313
 ESP_BLE_MESH_MODEL_OP_NET_KEY_LIST (C macro), 315
 ESP_BLE_MESH_MODEL_OP_NET_KEY_STATUS (C macro), 315
 ESP_BLE_MESH_MODEL_OP_NET_KEY_UPDATE (C macro), 315
 ESP_BLE_MESH_MODEL_OP_NETWORK_TRANSMIT_GET (C macro), 314
 ESP_BLE_MESH_MODEL_OP_NETWORK_TRANSMIT_SET (C macro), 315
 ESP_BLE_MESH_MODEL_OP_NETWORK_TRANSMIT_STATUS (C macro), 316
 ESP_BLE_MESH_MODEL_OP_NODE_IDENTITY_GET (C macro), 314
 ESP_BLE_MESH_MODEL_OP_NODE_IDENTITY_SET (C macro), 315
 ESP_BLE_MESH_MODEL_OP_NODE_IDENTITY_STATUS (C macro), 315
 ESP_BLE_MESH_MODEL_OP_NODE_RESET (C macro), 315
 ESP_BLE_MESH_MODEL_OP_NODE_RESET_STATUS (C macro), 316
 ESP_BLE_MESH_MODEL_OP_RELAY_GET (C macro), 313
 ESP_BLE_MESH_MODEL_OP_RELAY_SET (C macro), 314
 ESP_BLE_MESH_MODEL_OP_RELAY_STATUS (C macro), 315
 ESP_BLE_MESH_MODEL_OP_SCENE_DELETE (C macro), 320
 ESP_BLE_MESH_MODEL_OP_SCENE_DELETE_UNACK (C macro), 320
 ESP_BLE_MESH_MODEL_OP_SCENE_GET (C macro), 319
 ESP_BLE_MESH_MODEL_OP_SCENE_RECALL (C macro), 320
 ESP_BLE_MESH_MODEL_OP_SCENE_RECALL_UNACK (C macro), 320
 ESP_BLE_MESH_MODEL_OP_SCENE_REGISTER_GET (C macro), 320
 ESP_BLE_MESH_MODEL_OP_SCENE_REGISTER_STATUS (C macro), 320
 ESP_BLE_MESH_MODEL_OP_SCENE_STATUS (C macro), 320
 ESP_BLE_MESH_MODEL_OP_SCENE_STORE (C macro), 320
 ESP_BLE_MESH_MODEL_OP_SCENE_STORE_UNACK (C macro), 320
 ESP_BLE_MESH_MODEL_OP_SCHEDULER_ACT_GET (C macro), 320
 ESP_BLE_MESH_MODEL_OP_SCHEDULER_ACT_SET (C macro), 320
 ESP_BLE_MESH_MODEL_OP_SCHEDULER_ACT_SET_UNACK (C macro), 320
 ESP_BLE_MESH_MODEL_OP_SCHEDULER_ACT_STATUS (C macro), 320
 ESP_BLE_MESH_MODEL_OP_SCHEDULER_GET (C macro), 320
 ESP_BLE_MESH_MODEL_OP_SCHEDULER_STATUS (C macro), 320
 ESP_BLE_MESH_MODEL_OP_SENSOR_CADENCE_GET (C macro), 319
 ESP_BLE_MESH_MODEL_OP_SENSOR_CADENCE_SET (C macro), 319
 ESP_BLE_MESH_MODEL_OP_SENSOR_CADENCE_SET_UNACK (C macro), 319
 ESP_BLE_MESH_MODEL_OP_SENSOR_CADENCE_STATUS (C macro), 319
 ESP_BLE_MESH_MODEL_OP_SENSOR_COLUMN_GET (C macro), 319
 ESP_BLE_MESH_MODEL_OP_SENSOR_COLUMN_STATUS (C macro), 319
 ESP_BLE_MESH_MODEL_OP_SENSOR_DESCRIPTOR_GET (C macro), 319
 ESP_BLE_MESH_MODEL_OP_SENSOR_DESCRIPTOR_STATUS (C macro), 319
 ESP_BLE_MESH_MODEL_OP_SENSOR_GET (C macro), 319
 ESP_BLE_MESH_MODEL_OP_SENSOR_SERIES_GET (C macro), 319
 ESP_BLE_MESH_MODEL_OP_SENSOR_SERIES_STATUS (C macro), 319
 ESP_BLE_MESH_MODEL_OP_SENSOR_SETTING_GET (C macro), 319
 ESP_BLE_MESH_MODEL_OP_SENSOR_SETTING_SET (C macro), 319
 ESP_BLE_MESH_MODEL_OP_SENSOR_SETTING_SET_UNACK (C macro), 319
 ESP_BLE_MESH_MODEL_OP_SENSOR_SETTING_STATUS (C macro), 319
 ESP_BLE_MESH_MODEL_OP_SENSOR_SETTINGS_GET (C macro), 319
 ESP_BLE_MESH_MODEL_OP_SENSOR_SETTINGS_STATUS (C macro), 319
 ESP_BLE_MESH_MODEL_OP_SENSOR_STATUS (C macro), 319
 esp_ble_mesh_model_op_t (C++ class), 301
 esp_ble_mesh_model_op_t::min_len (C++ member), 302
 esp_ble_mesh_model_op_t::opcode (C++ member), 302
 esp_ble_mesh_model_op_t::param_cb (C++ member), 302
 ESP_BLE_MESH_MODEL_OP_TAI_UTC_DELTA_GET (C macro), 320

- (*C macro*), 319
- ESP_BLE_MESH_MODEL_OP_TAI.UTC_DELTA_SET (*C macro*), 319
- ESP_BLE_MESH_MODEL_OP_TAI.UTC_DELTA_STATUS (*C macro*), 319
- ESP_BLE_MESH_MODEL_OP_TIME_GET (*C macro*), 319
- ESP_BLE_MESH_MODEL_OP_TIME_ROLE_GET (*C macro*), 319
- ESP_BLE_MESH_MODEL_OP_TIME_ROLE_SET (*C macro*), 319
- ESP_BLE_MESH_MODEL_OP_TIME_ROLE_STATUS (*C macro*), 319
- ESP_BLE_MESH_MODEL_OP_TIME_SET (*C macro*), 319
- ESP_BLE_MESH_MODEL_OP_TIME_STATUS (*C macro*), 319
- ESP_BLE_MESH_MODEL_OP_TIME_ZONE_GET (*C macro*), 319
- ESP_BLE_MESH_MODEL_OP_TIME_ZONE_SET (*C macro*), 319
- ESP_BLE_MESH_MODEL_OP_TIME_ZONE_STATUS (*C macro*), 319
- ESP_BLE_MESH_MODEL_OP_VENDOR_MODEL_APP_GET (*C macro*), 314
- ESP_BLE_MESH_MODEL_OP_VENDOR_MODEL_APP_LIST (*C macro*), 316
- ESP_BLE_MESH_MODEL_OP_VENDOR_MODEL_SUB_GET (*C macro*), 314
- ESP_BLE_MESH_MODEL_OP_VENDOR_MODEL_SUB_LIST (*C macro*), 315
- ESP_BLE_MESH_MODEL_OPERATION_EVT (*C++ enumerator*), 330
- ESP_BLE_MESH_MODEL_PUB_DEFINE (*C macro*), 311
- esp_ble_mesh_model_pub_t (*C++ class*), 301
- esp_ble_mesh_model_pub_t::app_idx (*C++ member*), 301
- esp_ble_mesh_model_pub_t::count (*C++ member*), 301
- esp_ble_mesh_model_pub_t::cred (*C++ member*), 301
- esp_ble_mesh_model_pub_t::dev_role (*C++ member*), 301
- esp_ble_mesh_model_pub_t::fast_period (*C++ member*), 301
- esp_ble_mesh_model_pub_t::model (*C++ member*), 301
- esp_ble_mesh_model_pub_t::msg (*C++ member*), 301
- esp_ble_mesh_model_pub_t::period (*C++ member*), 301
- esp_ble_mesh_model_pub_t::period_div (*C++ member*), 301
- esp_ble_mesh_model_pub_t::period_start (*C++ member*), 301
- esp_ble_mesh_model_pub_t::publish_addr (*C++ member*), 301
- esp_ble_mesh_model_pub_t::retransmit (*C++ member*), 301
- esp_ble_mesh_model_pub_t::send_rel (*C++ member*), 301
- esp_ble_mesh_model_pub_t::timer (*C++ member*), 301
- esp_ble_mesh_model_pub_t::ttl (*C++ member*), 301
- esp_ble_mesh_model_pub_t::update (*C++ member*), 301
- esp_ble_mesh_model_publish (*C++ function*), 335
- ESP_BLE_MESH_MODEL_PUBLISH_COMP_EVT (*C++ enumerator*), 330
- ESP_BLE_MESH_MODEL_PUBLISH_UPDATE_EVT (*C++ enumerator*), 330
- ESP_BLE_MESH_MODEL_SCENE_CLI (*C macro*), 485
- ESP_BLE_MESH_MODEL_SCENE_SETUP_SRV (*C macro*), 486
- ESP_BLE_MESH_MODEL_SCENE_SRV (*C macro*), 485
- ESP_BLE_MESH_MODEL_SCHEDULER_CLI (*C macro*), 485
- ESP_BLE_MESH_MODEL_SCHEDULER_SETUP_SRV (*C macro*), 486
- ESP_BLE_MESH_MODEL_SCHEDULER_SRV (*C macro*), 486
- ESP_BLE_MESH_MODEL_SEND_COMP_EVT (*C++ enumerator*), 330
- ESP_BLE_MESH_MODEL_SENSOR_CLI (*C macro*), 464
- ESP_BLE_MESH_MODEL_SENSOR_SETUP_SRV (*C macro*), 464
- ESP_BLE_MESH_MODEL_SENSOR_SRV (*C macro*), 464
- ESP_BLE_MESH_MODEL_STATUS_CANNOT_SET_RANGE_MAX (*C macro*), 322
- ESP_BLE_MESH_MODEL_STATUS_CANNOT_SET_RANGE_MIN (*C macro*), 322
- ESP_BLE_MESH_MODEL_STATUS_SUCCESS (*C macro*), 322
- esp_ble_mesh_model_status_t (*C++ type*), 324
- esp_ble_mesh_model_subscribe_group_addr (*C++ function*), 332
- ESP_BLE_MESH_MODEL_SUBSCRIBE_GROUP_ADDR_COMP_EVT (*C++ enumerator*), 329
- esp_ble_mesh_model_t (*C++ type*), 323
- ESP_BLE_MESH_MODEL_TIME_CLI (*C macro*), 485
- ESP_BLE_MESH_MODEL_TIME_SETUP_SRV (*C macro*), 485
- ESP_BLE_MESH_MODEL_TIME_SRV (*C macro*), 485
- esp_ble_mesh_model_unsubscribe_group_addr (*C++ function*), 332
- ESP_BLE_MESH_MODEL_UNSUBSCRIBE_GROUP_ADDR_COMP_EVT

- (C++ enumerator), 329
- esp_ble_mesh_msg_ctx_t (C++ class), 302
- esp_ble_mesh_msg_ctx_t::addr (C++ member), 303
- esp_ble_mesh_msg_ctx_t::app_idx (C++ member), 303
- esp_ble_mesh_msg_ctx_t::model (C++ member), 303
- esp_ble_mesh_msg_ctx_t::net_idx (C++ member), 303
- esp_ble_mesh_msg_ctx_t::rcv_dst (C++ member), 303
- esp_ble_mesh_msg_ctx_t::rcv_op (C++ member), 303
- esp_ble_mesh_msg_ctx_t::rcv_rssi (C++ member), 303
- esp_ble_mesh_msg_ctx_t::rcv_ttl (C++ member), 303
- esp_ble_mesh_msg_ctx_t::send_rel (C++ member), 303
- esp_ble_mesh_msg_ctx_t::send_ttl (C++ member), 303
- esp_ble_mesh_msg_ctx_t::srv_send (C++ member), 303
- ESP_BLE_MESH_NET_PRIMARY (C macro), 309
- ESP_BLE_MESH_NO_FAULT (C macro), 408
- ESP_BLE_MESH_NO_INPUT (C++ enumerator), 324
- ESP_BLE_MESH_NO_LOAD_ERROR (C macro), 408
- ESP_BLE_MESH_NO_LOAD_WARNING (C macro), 408
- ESP_BLE_MESH_NO_OOB (C++ enumerator), 324
- ESP_BLE_MESH_NO_OUTPUT (C++ enumerator), 324
- esp_ble_mesh_node_add_local_app_key (C++ function), 333
- ESP_BLE_MESH_NODE_ADD_LOCAL_APP_KEY_COMPLETE_EVT (C++ enumerator), 326
- esp_ble_mesh_node_add_local_net_key (C++ function), 333
- ESP_BLE_MESH_NODE_ADD_LOCAL_NET_KEY_COMPLETE_EVT (C++ enumerator), 326
- esp_ble_mesh_node_bind_app_key_to_local_net_key (C++ function), 333
- ESP_BLE_MESH_NODE_BIND_APP_KEY_TO_MODEL_COMPLETE_EVT (C++ enumerator), 326
- esp_ble_mesh_node_get_local_app_key (C++ function), 332
- esp_ble_mesh_node_get_local_net_key (C++ function), 332
- ESP_BLE_MESH_NODE_IDENTITY_NOT_SUPPORTED (C macro), 309
- ESP_BLE_MESH_NODE_IDENTITY_RUNNING (C macro), 309
- ESP_BLE_MESH_NODE_IDENTITY_STOPPED (C macro), 309
- esp_ble_mesh_node_input_number (C++ function), 342
- esp_ble_mesh_node_input_string (C++ function), 342
- esp_ble_mesh_node_is_provisioned (C++ function), 341
- esp_ble_mesh_node_local_reset (C++ function), 336
- ESP_BLE_MESH_NODE_NAME_MAX_LEN (C macro), 308
- ESP_BLE_MESH_NODE_PROV_COMPLETE_EVT (C++ enumerator), 326
- esp_ble_mesh_node_prov_disable (C++ function), 341
- ESP_BLE_MESH_NODE_PROV_DISABLE_COMP_EVT (C++ enumerator), 326
- esp_ble_mesh_node_prov_enable (C++ function), 341
- ESP_BLE_MESH_NODE_PROV_ENABLE_COMP_EVT (C++ enumerator), 326
- ESP_BLE_MESH_NODE_PROV_INPUT_EVT (C++ enumerator), 326
- ESP_BLE_MESH_NODE_PROV_INPUT_NUMBER_COMP_EVT (C++ enumerator), 326
- ESP_BLE_MESH_NODE_PROV_INPUT_STRING_COMP_EVT (C++ enumerator), 326
- ESP_BLE_MESH_NODE_PROV_LINK_CLOSE_EVT (C++ enumerator), 326
- ESP_BLE_MESH_NODE_PROV_LINK_OPEN_EVT (C++ enumerator), 326
- ESP_BLE_MESH_NODE_PROV_OOB_PUB_KEY_EVT (C++ enumerator), 326
- ESP_BLE_MESH_NODE_PROV_OUTPUT_NUMBER_EVT (C++ enumerator), 326
- ESP_BLE_MESH_NODE_PROV_OUTPUT_STRING_EVT (C++ enumerator), 326
- ESP_BLE_MESH_NODE_PROV_RESET_EVT (C++ enumerator), 326
- ESP_BLE_MESH_NODE_PROV_SET_OOB_PUB_KEY_COMP_EVT (C++ enumerator), 326
- ESP_BLE_MESH_NODE_PROXY_GATT_DISABLE_COMP_EVT (C++ enumerator), 326
- ESP_BLE_MESH_NODE_PROXY_GATT_ENABLE_COMP_EVT (C++ enumerator), 326
- ESP_BLE_MESH_NODE_PROXY_IDENTITY_ENABLE_COMP_EVT (C++ enumerator), 326
- esp_ble_mesh_node_set_oob_pub_key (C++ function), 341
- ESP_BLE_MESH_NODE_SET_UNPROV_DEV_NAME_COMP_EVT (C++ enumerator), 326
- esp_ble_mesh_node_t (C++ class), 304
- esp_ble_mesh_node_t::addr (C++ member), 304
- esp_ble_mesh_node_t::addr_type (C++ member), 304
- esp_ble_mesh_node_t::comp_data (C++ member), 305
- esp_ble_mesh_node_t::comp_length (C++ member), 305
- esp_ble_mesh_node_t::dev_key (C++ member), 305

- esp_ble_mesh_node_t::dev_uuid (C++ member), 304
 esp_ble_mesh_node_t::element_num (C++ member), 305
 esp_ble_mesh_node_t::flags (C++ member), 305
 esp_ble_mesh_node_t::iv_index (C++ member), 305
 esp_ble_mesh_node_t::name (C++ member), 305
 esp_ble_mesh_node_t::net_idx (C++ member), 305
 esp_ble_mesh_node_t::oob_info (C++ member), 305
 esp_ble_mesh_node_t::unicast_addr (C++ member), 305
 ESP_BLE_MESH_OCTET16_LEN (C macro), 308
 esp_ble_mesh_octet16_t (C++ type), 323
 ESP_BLE_MESH_OCTET8_LEN (C macro), 308
 esp_ble_mesh_octet8_t (C++ type), 323
 esp_ble_mesh_oob_method_t (C++ enum), 324
 esp_ble_mesh_opcode_config_client_get_t (C++ type), 323
 esp_ble_mesh_opcode_config_client_set_t (C++ type), 323
 esp_ble_mesh_opcode_config_status_t (C++ type), 323
 esp_ble_mesh_opcode_health_client_get_t (C++ type), 323
 esp_ble_mesh_opcode_health_client_set_t (C++ type), 323
 esp_ble_mesh_opcode_t (C++ type), 324
 esp_ble_mesh_output_action_t (C++ enum), 324
 ESP_BLE_MESH_OUTPUT_OOB (C++ enumerator), 324
 ESP_BLE_MESH_OVERFLOW_ERROR (C macro), 409
 ESP_BLE_MESH_OVERFLOW_WARNING (C macro), 409
 ESP_BLE_MESH_OVERHEAT_ERROR (C macro), 408
 ESP_BLE_MESH_OVERHEAT_WARNING (C macro), 408
 ESP_BLE_MESH_OVERLOAD_ERROR (C macro), 408
 ESP_BLE_MESH_OVERLOAD_WARNING (C macro), 408
 ESP_BLE_MESH_POWER_SUPPLY_INTERRUPTED_ERROR (C macro), 408
 ESP_BLE_MESH_POWER_SUPPLY_INTERRUPTED_WARNING (C macro), 408
 ESP_BLE_MESH_PROV (C macro), 311
 ESP_BLE_MESH_PROV_ADV (C++ enumerator), 325
 esp_ble_mesh_prov_adv_cb_t (C++ type), 345
 esp_ble_mesh_prov_bearer_t (C++ enum), 325
 esp_ble_mesh_prov_cb_event_t (C++ enum), 325
 esp_ble_mesh_prov_cb_param_t (C++ union), 278
 esp_ble_mesh_prov_cb_param_t::ble_mesh_deinit_mes (C++ class), 282
 esp_ble_mesh_prov_cb_param_t::ble_mesh_deinit_mes (C++ member), 283
 esp_ble_mesh_prov_cb_param_t::ble_mesh_friend_fri (C++ class), 283
 esp_ble_mesh_prov_cb_param_t::ble_mesh_friend_fri (C++ member), 283
 esp_ble_mesh_prov_cb_param_t::ble_mesh_friend_fri (C++ member), 283
 esp_ble_mesh_prov_cb_param_t::ble_mesh_friend_fri (C++ class), 283
 esp_ble_mesh_prov_cb_param_t::ble_mesh_friend_fri (C++ member), 283
 esp_ble_mesh_prov_cb_param_t::ble_mesh_friend_fri (C++ member), 283
 esp_ble_mesh_prov_cb_param_t::ble_mesh_heartbeat (C++ class), 283
 esp_ble_mesh_prov_cb_param_t::ble_mesh_heartbeat (C++ member), 283
 esp_ble_mesh_prov_cb_param_t::ble_mesh_heartbeat (C++ member), 283
 esp_ble_mesh_prov_cb_param_t::ble_mesh_heartbeat (C++ class), 283
 esp_ble_mesh_prov_cb_param_t::ble_mesh_heartbeat (C++ member), 283
 esp_ble_mesh_prov_cb_param_t::ble_mesh_input_evt (C++ class), 283
 esp_ble_mesh_prov_cb_param_t::ble_mesh_input_evt (C++ member), 284
 esp_ble_mesh_prov_cb_param_t::ble_mesh_input_evt (C++ member), 284
 esp_ble_mesh_prov_cb_param_t::ble_mesh_input_evt (C++ class), 284
 esp_ble_mesh_prov_cb_param_t::ble_mesh_input_num (C++ member), 284
 esp_ble_mesh_prov_cb_param_t::ble_mesh_input_num (C++ member), 284
 esp_ble_mesh_prov_cb_param_t::ble_mesh_input_stri (C++ class), 284
 esp_ble_mesh_prov_cb_param_t::ble_mesh_input_stri (C++ member), 284
 esp_ble_mesh_prov_cb_param_t::ble_mesh_input_stri (C++ member), 284
 esp_ble_mesh_prov_cb_param_t::ble_mesh_link_close (C++ class), 284
 esp_ble_mesh_prov_cb_param_t::ble_mesh_link_close (C++ member), 284
 esp_ble_mesh_prov_cb_param_t::ble_mesh_link_close (C++ member), 284
 esp_ble_mesh_prov_cb_param_t::ble_mesh_link_open (C++ class), 284
 esp_ble_mesh_prov_cb_param_t::ble_mesh_link_open (C++ member), 284
 esp_ble_mesh_prov_cb_param_t::ble_mesh_link_open (C++ member), 284
 esp_ble_mesh_prov_cb_param_t::ble_mesh_lpn_disabl (C++ class), 284
 esp_ble_mesh_prov_cb_param_t::ble_mesh_lpn_disabl (C++ member), 284
 esp_ble_mesh_prov_cb_param_t::ble_mesh_lpn_enable (C++ class), 284
 esp_ble_mesh_prov_cb_param_t::ble_mesh_lpn_enable (C++ member), 285
 esp_ble_mesh_prov_cb_param_t::ble_mesh_lpn_friend (C++ class), 285
 esp_ble_mesh_prov_cb_param_t::ble_mesh_lpn_friend (C++ member), 285
 esp_ble_mesh_prov_cb_param_t::ble_mesh_lpn_friend (C++ member), 285
 esp_ble_mesh_prov_cb_param_t::ble_mesh_lpn_friend (C++ member), 285

- (C++ member), 278
- `esp_ble_mesh_prov_cb_param_t::node_provisioner_prov_in` (C++ member), 278
- `esp_ble_mesh_prov_cb_param_t::node_provisioner_prov_li` (C++ member), 278
- `esp_ble_mesh_prov_cb_param_t::node_provisioner_prov_li` (C++ member), 279
- `esp_ble_mesh_prov_cb_param_t::node_provisioner_prov_ou` (C++ member), 278
- `esp_ble_mesh_prov_cb_param_t::node_provisioner_prov_re` (C++ member), 278
- `esp_ble_mesh_prov_cb_param_t::node_set_name` (C++ member), 278
- `esp_ble_mesh_prov_cb_param_t::op` (C++ member), 280
- `esp_ble_mesh_prov_cb_param_t::prov_register` (C++ member), 278
- `esp_ble_mesh_prov_cb_param_t::provisioner_set_dev` (C++ member), 280
- `esp_ble_mesh_prov_cb_param_t::provisioner_set_he` (C++ member), 280
- `esp_ble_mesh_prov_cb_param_t::provisioner_set_he` (C++ member), 279
- `esp_ble_mesh_prov_cb_param_t::provisioner_set_nod` (C++ member), 280
- `esp_ble_mesh_prov_cb_param_t::provisioner_set_pri` (C++ member), 281
- `esp_ble_mesh_prov_cb_param_t::provisioner_set_pro` (C++ member), 281
- `esp_ble_mesh_prov_cb_param_t::provisioner_set_sta` (C++ member), 279
- `esp_ble_mesh_prov_cb_param_t::provisioner_store_n` (C++ member), 280
- `esp_ble_mesh_prov_cb_param_t::provisioner_update` (C++ member), 280
- `esp_ble_mesh_prov_cb_param_t::provisioner_update` (C++ member), 281
- `esp_ble_mesh_prov_cb_param_t::provisioner_update` (C++ member), 281
- `esp_ble_mesh_prov_cb_param_t::provisioner_update` (C++ member), 282
- `esp_ble_mesh_prov_cb_param_t::provisioner_update` (C++ member), 282
- `esp_ble_mesh_prov_cb_param_t::provisioner_update` (C++ member), 282
- `esp_ble_mesh_prov_cb_param_t::provisioner_update` (C++ member), 282
- `esp_ble_mesh_prov_cb_param_t::proxy_client_add_fi` (C++ member), 281
- `esp_ble_mesh_prov_cb_param_t::proxy_client_connec` (C++ member), 281
- `esp_ble_mesh_prov_cb_param_t::proxy_client_connec` (C++ member), 280
- `esp_ble_mesh_prov_cb_param_t::proxy_client_discon` (C++ member), 281
- `esp_ble_mesh_prov_cb_param_t::proxy_client_discon` (C++ member), 281
- `esp_ble_mesh_prov_cb_param_t::proxy_client_recv_a` (C++ member), 279
- `esp_ble_mesh_prov_cb_param_t::proxy_client_recv_f` (C++ member), 279
- `esp_ble_mesh_prov_cb_param_t::proxy_client_remove` (C++ member), 279
- `esp_ble_mesh_prov_cb_param_t::proxy_client_set_fi` (C++ member), 279
- `esp_ble_mesh_prov_cb_param_t::rssi` (C++ member), 279
- `esp_ble_mesh_prov_cb_param_t::rx_ttl`

- (C++ member), 281
- esp_ble_mesh_prov_cb_param_t::set_fast_prov_a(C++ member), 310
- (C++ member), 281
- esp_ble_mesh_prov_cb_param_t::set_fast_prov_b(C++ member), 281
- (C++ member), 281
- esp_ble_mesh_prov_cb_param_t::type (C++ member), 280
- (C++ member), 281
- esp_ble_mesh_prov_cb_param_t::uid (C++ member), 281
- (C++ member), 281
- esp_ble_mesh_prov_cb_param_t::[anonymous] (C++ enum), 283
- esp_ble_mesh_prov_cb_t (C++ type), 345
- esp_ble_mesh_prov_data_info_t (C++ class), 304
- esp_ble_mesh_prov_data_info_t::flag (C++ member), 304
- esp_ble_mesh_prov_data_info_t::flags (C++ member), 304
- esp_ble_mesh_prov_data_info_t::iv_index (C++ member), 304
- esp_ble_mesh_prov_data_info_t::net_idx (C++ member), 304
- ESP_BLE_MESH_PROV_EVT_MAX (C++ enumerator), 329
- ESP_BLE_MESH_PROV_GATT (C++ enumerator), 325
- ESP_BLE_MESH_PROV_INPUT_OOB_MAX_LEN (C macro), 310
- ESP_BLE_MESH_PROV_OOB_2D_CODE (C++ enumerator), 325
- ESP_BLE_MESH_PROV_OOB_BAR_CODE (C++ enumerator), 325
- ESP_BLE_MESH_PROV_OOB_IN_BOX (C++ enumerator), 325
- ESP_BLE_MESH_PROV_OOB_IN_MANUAL (C++ enumerator), 325
- esp_ble_mesh_prov_oob_info_t (C++ enum), 325
- ESP_BLE_MESH_PROV_OOB_NFC (C++ enumerator), 325
- ESP_BLE_MESH_PROV_OOB_NUMBER (C++ enumerator), 325
- ESP_BLE_MESH_PROV_OOB_ON_BOX (C++ enumerator), 325
- ESP_BLE_MESH_PROV_OOB_ON_DEV (C++ enumerator), 325
- ESP_BLE_MESH_PROV_OOB_ON_PAPER (C++ enumerator), 325
- ESP_BLE_MESH_PROV_OOB_OTHER (C++ enumerator), 325
- ESP_BLE_MESH_PROV_OOB_STRING (C++ enumerator), 325
- ESP_BLE_MESH_PROV_OOB_URI (C++ enumerator), 325
- ESP_BLE_MESH_PROV_OUTPUT_OOB_MAX_LEN (C macro), 310
- ESP_BLE_MESH_PROV_REGISTER_COMP_EVT (C++ enumerator), 325
- ESP_BLE_MESH_PROV_STATIC_OOB_MAX_LEN (C macro), 310
- esp_ble_mesh_prov_t (C++ class), 303
- esp_ble_mesh_provisioner_add_local_app_key (C++ function), 337
- ESP_BLE_MESH_PROVISIONER_ADD_LOCAL_APP_KEY_COMP_EVT (C++ enumerator), 327
- esp_ble_mesh_provisioner_add_local_net_key (C++ function), 338
- ESP_BLE_MESH_PROVISIONER_ADD_LOCAL_NET_KEY_COMP_EVT (C++ enumerator), 327
- esp_ble_mesh_provisioner_add_unprov_dev (C++ function), 343
- ESP_BLE_MESH_PROVISIONER_ADD_UNPROV_DEV_COMP_EVT (C++ enumerator), 327
- esp_ble_mesh_provisioner_bind_app_key_to_local_model (C++ function), 338
- ESP_BLE_MESH_PROVISIONER_BIND_APP_KEY_TO_MODEL_COMP_EVT (C++ enumerator), 327
- esp_ble_mesh_provisioner_close_settings_with_index (C++ function), 339
- ESP_BLE_MESH_PROVISIONER_CLOSE_SETTINGS_WITH_INDEX (C++ enumerator), 328
- esp_ble_mesh_provisioner_close_settings_with_uid (C++ function), 340
- ESP_BLE_MESH_PROVISIONER_CLOSE_SETTINGS_WITH_UID (C++ enumerator), 328
- esp_ble_mesh_provisioner_delete_dev (C++ function), 344
- ESP_BLE_MESH_PROVISIONER_DELETE_DEV_COMP_EVT (C++ enumerator), 327
- esp_ble_mesh_provisioner_delete_node_with_addr (C++ function), 337
- ESP_BLE_MESH_PROVISIONER_DELETE_NODE_WITH_ADDR_COMP_EVT (C++ enumerator), 328
- esp_ble_mesh_provisioner_delete_node_with_uuid (C++ function), 337
- ESP_BLE_MESH_PROVISIONER_DELETE_NODE_WITH_UUID_COMP_EVT (C++ enumerator), 328
- esp_ble_mesh_provisioner_delete_settings_with_index (C++ function), 340
- ESP_BLE_MESH_PROVISIONER_DELETE_SETTINGS_WITH_INDEX (C++ enumerator), 328
- esp_ble_mesh_provisioner_delete_settings_with_uid (C++ function), 340
- ESP_BLE_MESH_PROVISIONER_DELETE_SETTINGS_WITH_UID (C++ enumerator), 328
- esp_ble_mesh_provisioner_direct_erase_settings (C++ function), 339
- ESP_BLE_MESH_PROVISIONER_DRIECT_ERASE_SETTINGS_COMP_EVT (C++ enumerator), 328
- ESP_BLE_MESH_PROVISIONER_ENABLE_HEARTBEAT_RECV_COMP_EVT (C++ enumerator), 328
- esp_ble_mesh_provisioner_get_free_settings_count (C++ function), 340
- esp_ble_mesh_provisioner_get_local_app_key (C++ function), 337
- esp_ble_mesh_provisioner_get_local_net_key

(C++ function), 338
 esp_ble_mesh_provisioner_get_node_index (C++ function), 336
 esp_ble_mesh_provisioner_get_node_name (C++ function), 336
 esp_ble_mesh_provisioner_get_node_table (C++ function), 337
 esp_ble_mesh_provisioner_get_node_with_espid (C++ function), 336
 esp_ble_mesh_provisioner_get_node_with_espid_and_name (C++ function), 336
 esp_ble_mesh_provisioner_get_prov_node_espid (C++ function), 337
 esp_ble_mesh_provisioner_get_settings_index (C++ function), 340
 esp_ble_mesh_provisioner_get_settings_value (C++ function), 340
 esp_ble_mesh_provisioner_input_number (C++ function), 342
 esp_ble_mesh_provisioner_input_string (C++ function), 342
 esp_ble_mesh_provisioner_open_settings_index (C++ function), 339
 ESP_BLE_MESH_PROVISIONER_OPEN_SETTINGS_INDEX (C++ enumerator), 328
 esp_ble_mesh_provisioner_open_settings_value (C++ function), 339
 ESP_BLE_MESH_PROVISIONER_OPEN_SETTINGS_VALUE (C++ enumerator), 328
 esp_ble_mesh_provisioner_open_settings_value_espid (C++ function), 339
 ESP_BLE_MESH_PROVISIONER_OPEN_SETTINGS_VALUE_ESPID (C++ enumerator), 327
 ESP_BLE_MESH_PROVISIONER_PROV_COMPLETE (C++ enumerator), 327
 esp_ble_mesh_provisioner_prov_device_wid (C++ function), 343
 esp_ble_mesh_provisioner_prov_disable (C++ function), 343
 ESP_BLE_MESH_PROVISIONER_PROV_DISABLE_COMPLETE (C++ enumerator), 326
 esp_ble_mesh_provisioner_prov_enable (C++ function), 342
 ESP_BLE_MESH_PROVISIONER_PROV_ENABLE_COMPLETE (C++ enumerator), 326
 ESP_BLE_MESH_PROVISIONER_PROV_INPUT_EVT (C++ enumerator), 327
 ESP_BLE_MESH_PROVISIONER_PROV_INPUT_NUMBER_COMPLETE (C++ enumerator), 327
 ESP_BLE_MESH_PROVISIONER_PROV_INPUT_STRING_COMPLETE (C++ enumerator), 327
 ESP_BLE_MESH_PROVISIONER_PROV_LINK_CLOSE_EVT (C++ enumerator), 327
 ESP_BLE_MESH_PROVISIONER_PROV_LINK_OPEN_EVT (C++ enumerator), 327
 ESP_BLE_MESH_PROVISIONER_PROV_OUTPUT_EVT (C++ enumerator), 327
 ESP_BLE_MESH_PROVISIONER_PROV_READ_OOB_ESP_KEY_MESH (C++ enumerator), 327
 (C++ enumerator), 327
 ESP_BLE_MESH_PROVISIONER_PROV_READ_OOB_PUB_KEY_EVT (C++ enumerator), 327
 esp_ble_mesh_provisioner_read_oob_pub_key (C++ function), 342
 esp_ble_mesh_provisioner_rcv_heartbeat (C++ function), 338
 ESP_BLE_MESH_PROVISIONER_RECV_HEARTBEAT_MESSAGE_TYPE (C++ enumerator), 328
 esp_ble_mesh_provisioner_rcv_unprov_adv_pkt_evt (C++ enumerator), 326
 esp_ble_mesh_provisioner_set_dev_uuid_match (C++ function), 344
 ESP_BLE_MESH_PROVISIONER_SET_DEV_UUID_MATCH_COMPLETE (C++ enumerator), 327
 esp_ble_mesh_provisioner_set_heartbeat_filter_info (C++ function), 339
 ESP_BLE_MESH_PROVISIONER_SET_HEARTBEAT_FILTER_INFO (C++ enumerator), 328
 esp_ble_mesh_provisioner_set_heartbeat_filter_type (C++ function), 338
 ESP_BLE_MESH_PROVISIONER_SET_HEARTBEAT_FILTER_TYPE (C++ enumerator), 328
 esp_ble_mesh_provisioner_set_node_name (C++ function), 336
 ESP_BLE_MESH_PROVISIONER_SET_NODE_NAME_COMPLETE_EVT (C++ enumerator), 327
 esp_ble_mesh_provisioner_set_primary_elem_addr (C++ function), 345
 ESP_BLE_MESH_PROVISIONER_SET_PRIMARY_ELEM_ADDR_COMPLETE (C++ enumerator), 327
 esp_ble_mesh_provisioner_set_prov_data_info (C++ function), 344
 ESP_BLE_MESH_PROVISIONER_SET_PROV_DATA_INFO_COMPLETE (C++ enumerator), 327
 esp_ble_mesh_provisioner_set_static_oob_value (C++ function), 344
 ESP_BLE_MESH_PROVISIONER_SET_STATIC_OOB_VALUE_COMPLETE (C++ enumerator), 327
 esp_ble_mesh_provisioner_store_node_comp_data (C++ function), 336
 ESP_BLE_MESH_PROVISIONER_STORE_NODE_COMP_DATA_COMPLETE (C++ enumerator), 327
 esp_ble_mesh_provisioner_update_local_app_key (C++ function), 337
 ESP_BLE_MESH_PROVISIONER_UPDATE_LOCAL_APP_KEY_COMPLETE (C++ enumerator), 327
 esp_ble_mesh_provisioner_update_local_net_key (C++ function), 338
 ESP_BLE_MESH_PROVISIONER_UPDATE_LOCAL_NET_KEY_COMPLETE (C++ enumerator), 327
 esp_ble_mesh_proxy_client_add_filter_addr (C++ function), 346
 ESP_BLE_MESH_PROXY_CLIENT_ADD_FILTER_ADDR_COMPLETE_EVT (C++ enumerator), 329
 esp_ble_mesh_proxy_client_connect (C++ function), 346
 ESP_BLE_MESH_PROXY_CLIENT_CONNECT_COMPLETE_EVT (C++ enumerator), 329

- (C++ enumerator), 329
- ESP_BLE_MESH_PROXY_CLIENT_CONNECTED_EVT (C++ enumerator), 329
- esp_ble_mesh_proxy_client_disconnect (C++ function), 346
- ESP_BLE_MESH_PROXY_CLIENT_DISCONNECT_COMPLETE (C++ enumerator), 329
- ESP_BLE_MESH_PROXY_CLIENT_DISCONNECTED_EVT (C++ enumerator), 329
- ESP_BLE_MESH_PROXY_CLIENT_RECV_ADV_PKT_EVT (C++ enumerator), 329
- ESP_BLE_MESH_PROXY_CLIENT_RECV_FILTER_STATUS_EVT (C++ enumerator), 329
- esp_ble_mesh_proxy_client_remove_filter_status (C++ function), 347
- ESP_BLE_MESH_PROXY_CLIENT_REMOVE_FILTER_STATUS_EVT (C++ enumerator), 329
- esp_ble_mesh_proxy_client_set_filter_type (C++ function), 346
- ESP_BLE_MESH_PROXY_CLIENT_SET_FILTER_TYPE_COMPLETE (C++ enumerator), 329
- esp_ble_mesh_proxy_filter_type_t (C++ enum), 325
- esp_ble_mesh_proxy_gatt_disable (C++ function), 346
- esp_ble_mesh_proxy_gatt_enable (C++ function), 346
- esp_ble_mesh_proxy_identity_enable (C++ function), 346
- ESP_BLE_MESH_PUBLISH_TRANSMIT (C macro), 310
- ESP_BLE_MESH_PUSH (C++ enumerator), 325
- esp_ble_mesh_register_config_client_callback (C++ function), 347
- esp_ble_mesh_register_config_server_callback (C++ function), 347
- esp_ble_mesh_register_custom_model_callback (C++ function), 334
- esp_ble_mesh_register_generic_client_callback (C++ function), 371
- esp_ble_mesh_register_generic_server_callback (C++ function), 371
- esp_ble_mesh_register_health_client_callback (C++ function), 402
- esp_ble_mesh_register_health_server_callback (C++ function), 402
- esp_ble_mesh_register_light_client_callback (C++ function), 410
- esp_ble_mesh_register_lighting_server_callback (C++ function), 411
- esp_ble_mesh_register_prov_callback (C++ function), 341
- esp_ble_mesh_register_sensor_client_callback (C++ function), 452
- esp_ble_mesh_register_sensor_server_callback (C++ function), 453
- esp_ble_mesh_register_time_scene_client_callback (C++ function), 467
- esp_ble_mesh_register_time_scene_server_callback (C++ function), 468
- ESP_BLE_MESH_RELAY_DISABLED (C macro), 309
- ESP_BLE_MESH_RELAY_ENABLED (C macro), 309
- ESP_BLE_MESH_RELAY_NOT_SUPPORTED (C macro), 309
- ESP_BLE_MESH_SAMPLE_FUNC_ACCUMULATED (C++ enumerator), 467
- ESP_BLE_MESH_SAMPLE_FUNC_ARITHMETIC_MEAN (C++ enumerator), 467
- ESP_BLE_MESH_SAMPLE_FUNC_COUNT (C++ enumerator), 467
- ESP_BLE_MESH_SAMPLE_FUNC_INSTANTANEOUS (C++ enumerator), 467
- ESP_BLE_MESH_SAMPLE_FUNC_MAXIMUM (C++ enumerator), 467
- ESP_BLE_MESH_SAMPLE_FUNC_MINIMUM (C++ enumerator), 467
- ESP_BLE_MESH_SAMPLE_FUNC_RMS (C++ enumerator), 467
- ESP_BLE_MESH_SAMPLE_FUNC_UNSPECIFIED (C++ enumerator), 467
- esp_ble_mesh_scene_delete_t (C++ class), 472
- esp_ble_mesh_scene_delete_t::scene_number (C++ member), 472
- ESP_BLE_MESH_SCENE_NOT_FOUND (C macro), 488
- ESP_BLE_MESH_SCENE_NUMBER_LEN (C macro), 486
- esp_ble_mesh_scene_recall_t (C++ class), 472
- esp_ble_mesh_scene_recall_t::delay (C++ member), 472
- esp_ble_mesh_scene_recall_t::open (C++ member), 472
- esp_ble_mesh_scene_recall_t::scene_number (C++ member), 472
- esp_ble_mesh_scene_recall_t::tid (C++ member), 472
- esp_ble_mesh_scene_recall_t::trans_time (C++ member), 472
- ESP_BLE_MESH_SCENE_REG_FULL (C macro), 487
- esp_ble_mesh_scene_register_status_cb_t (C++ class), 474
- esp_ble_mesh_scene_register_status_cb_t::current (C++ member), 474
- esp_ble_mesh_scene_register_status_cb_t::scenes (C++ member), 474
- esp_ble_mesh_scene_register_status_cb_t::status (C++ member), 474
- esp_ble_mesh_scene_register_t (C++ class), 476
- esp_ble_mesh_scene_register_t::scene_number (C++ member), 477
- esp_ble_mesh_scene_register_t::scene_type (C++ member), 477

(C++ member), 477
 esp_ble_mesh_scene_register_t::scene_value (C++ member), 477
 esp_ble_mesh_scene_setup_srv_t (C++ class), 478
 esp_ble_mesh_scene_setup_srv_t::model (C++ member), 478
 esp_ble_mesh_scene_setup_srv_t::rsp_ctrl (C++ member), 478
 esp_ble_mesh_scene_setup_srv_t::state (C++ member), 478
 esp_ble_mesh_scene_srv_t (C++ class), 478
 esp_ble_mesh_scene_srv_t::last (C++ member), 478
 esp_ble_mesh_scene_srv_t::model (C++ member), 478
 esp_ble_mesh_scene_srv_t::rsp_ctrl (C++ member), 478
 esp_ble_mesh_scene_srv_t::state (C++ member), 478
 esp_ble_mesh_scene_srv_t::transition (C++ member), 478
 esp_ble_mesh_scene_status_cb_t (C++ class), 474
 esp_ble_mesh_scene_status_cb_t::current_scene (C++ member), 474
 esp_ble_mesh_scene_status_cb_t::open (C++ member), 474
 esp_ble_mesh_scene_status_cb_t::remain_time (C++ member), 474
 esp_ble_mesh_scene_status_cb_t::status_code (C++ member), 474
 esp_ble_mesh_scene_status_cb_t::target_scene (C++ member), 474
 esp_ble_mesh_scene_store_t (C++ class), 471
 esp_ble_mesh_scene_store_t::scene_number (C++ member), 472
 ESP_BLE_MESH_SCENE_SUCCESS (C macro), 487
 esp_ble_mesh_scenes_state_t (C++ class), 477
 esp_ble_mesh_scenes_state_t::current_scene (C++ member), 477
 esp_ble_mesh_scenes_state_t::in_progress (C++ member), 478
 esp_ble_mesh_scenes_state_t::scene_count (C++ member), 477
 esp_ble_mesh_scenes_state_t::scenes (C++ member), 477
 esp_ble_mesh_scenes_state_t::status_code (C++ member), 478
 esp_ble_mesh_scenes_state_t::target_scene (C++ member), 477
 ESP_BLE_MESH_SCHEDULE_ACT_NO_ACTION (C macro), 487
 ESP_BLE_MESH_SCHEDULE_ACT_SCENE_RECALL (C macro), 487
 ESP_BLE_MESH_SCHEDULE_ACT_TURN_OFF (C macro), 487
 ESP_BLE_MESH_SCHEDULE_ACT_TURN_ON (C macro), 487
 ESP_BLE_MESH_SCHEDULE_DAY_ANY_DAY (C macro), 487
 ESP_BLE_MESH_SCHEDULE_ENTRY_MAX_INDEX (C macro), 487
 ESP_BLE_MESH_SCHEDULE_HOUR_ANY_HOUR (C macro), 487
 ESP_BLE_MESH_SCHEDULE_HOUR_ONCE_A_DAY (C macro), 487
 esp_ble_mesh_schedule_register_t (C++ class), 478
 esp_ble_mesh_schedule_register_t::action (C++ member), 479
 esp_ble_mesh_schedule_register_t::day (C++ member), 478
 esp_ble_mesh_schedule_register_t::day_of_week (C++ member), 479
 esp_ble_mesh_schedule_register_t::hour (C++ member), 478
 esp_ble_mesh_schedule_register_t::in_use (C++ member), 478
 esp_ble_mesh_schedule_register_t::minute (C++ member), 478
 esp_ble_mesh_schedule_register_t::month (C++ member), 478
 esp_ble_mesh_schedule_register_t::scene_number (C++ member), 479
 esp_ble_mesh_schedule_register_t::second (C++ member), 479
 esp_ble_mesh_schedule_register_t::trans_time (C++ member), 478
 esp_ble_mesh_schedule_register_t::year (C++ member), 478
 ESP_BLE_MESH_SCHEDULE_SCENE_NO_SCENE (C macro), 487
 ESP_BLE_MESH_SCHEDULE_SEC_ANY_OF_HOUR (C macro), 487
 ESP_BLE_MESH_SCHEDULE_SEC_ANY_OF_MIN (C macro), 487
 ESP_BLE_MESH_SCHEDULE_SEC EVERY 15_MIN (C macro), 487
 ESP_BLE_MESH_SCHEDULE_SEC EVERY 15_SEC (C macro), 487
 ESP_BLE_MESH_SCHEDULE_SEC EVERY 20_MIN (C macro), 487
 ESP_BLE_MESH_SCHEDULE_SEC EVERY 20_SEC (C macro), 487
 ESP_BLE_MESH_SCHEDULE_SEC ONCE AN HOUR (C macro), 487
 ESP_BLE_MESH_SCHEDULE_SEC ONCE AN MIN (C macro), 487
 ESP_BLE_MESH_SCHEDULE_YEAR_ANY_YEAR (C macro), 487
 esp_ble_mesh_scheduler_act_get_t (C++ class), 472
 esp_ble_mesh_scheduler_act_get_t::index

(C++ member), 472

esp_ble_mesh_scheduler_act_set_t (C++ class), 472

esp_ble_mesh_scheduler_act_set_t::action (C++ member), 473

esp_ble_mesh_scheduler_act_set_t::day (C++ member), 472

esp_ble_mesh_scheduler_act_set_t::day_of_week (C++ member), 473

esp_ble_mesh_scheduler_act_set_t::hour (C++ member), 472

esp_ble_mesh_scheduler_act_set_t::index (C++ member), 472

esp_ble_mesh_scheduler_act_set_t::minutes (C++ member), 472

esp_ble_mesh_scheduler_act_set_t::month (C++ member), 472

esp_ble_mesh_scheduler_act_set_t::scene (C++ member), 473

esp_ble_mesh_scheduler_act_set_t::second (C++ member), 473

esp_ble_mesh_scheduler_act_set_t::trans_time (C++ member), 473

esp_ble_mesh_scheduler_act_set_t::year (C++ member), 472

esp_ble_mesh_scheduler_act_status_cb_t (C++ class), 474

esp_ble_mesh_scheduler_act_status_cb_t::action (C++ member), 475

esp_ble_mesh_scheduler_act_status_cb_t::day (C++ member), 475

esp_ble_mesh_scheduler_act_status_cb_t::day_of_week (C++ member), 475

esp_ble_mesh_scheduler_act_status_cb_t::hour (C++ member), 475

esp_ble_mesh_scheduler_act_status_cb_t::index (C++ member), 475

esp_ble_mesh_scheduler_act_status_cb_t::minutes (C++ member), 475

esp_ble_mesh_scheduler_act_status_cb_t::month (C++ member), 475

esp_ble_mesh_scheduler_act_status_cb_t::scene (C++ member), 475

esp_ble_mesh_scheduler_act_status_cb_t::second (C++ member), 475

esp_ble_mesh_scheduler_act_status_cb_t::trans_time (C++ member), 475

esp_ble_mesh_scheduler_act_status_cb_t::year (C++ member), 475

esp_ble_mesh_scheduler_setup_srv_t (C++ class), 479

esp_ble_mesh_scheduler_setup_srv_t::model (C++ member), 479

esp_ble_mesh_scheduler_setup_srv_t::rsp_ctrl (C++ member), 479

esp_ble_mesh_scheduler_setup_srv_t::state (C++ member), 479

esp_ble_mesh_scheduler_srv_t (C++ class), 479

esp_ble_mesh_scheduler_srv_t::model (C++ member), 479

esp_ble_mesh_scheduler_srv_t::rsp_ctrl (C++ member), 479

esp_ble_mesh_scheduler_srv_t::state (C++ member), 479

esp_ble_mesh_scheduler_state_t (C++ class), 479

esp_ble_mesh_scheduler_state_t::schedule_count (C++ member), 479

esp_ble_mesh_scheduler_state_t::schedules (C++ member), 479

esp_ble_mesh_scheduler_status_cb_t (C++ class), 474

esp_ble_mesh_scheduler_status_cb_t::schedules (C++ member), 474

ESP_BLE_MESH_SCHEDULE_INTERVAL (C macro), 308

ESP_BLE_MESH_SELF_TEST_ERROR (C macro), 409

ESP_BLE_MESH_SELF_TEST_WARNING (C macro), 409

esp_ble_mesh_sensor_cadence_get_t (C++ class), 455

esp_ble_mesh_sensor_cadence_get_t::property_id (C++ member), 455

esp_ble_mesh_sensor_cadence_set_t (C++ class), 455

esp_ble_mesh_sensor_cadence_set_t::fast_cadence_high (C++ member), 455

esp_ble_mesh_sensor_cadence_set_t::fast_cadence_low (C++ member), 455

esp_ble_mesh_sensor_cadence_set_t::fast_cadence_property_id (C++ member), 455

esp_ble_mesh_sensor_cadence_set_t::property_id (C++ member), 455

esp_ble_mesh_sensor_cadence_set_t::status_min_interval (C++ member), 455

esp_ble_mesh_sensor_cadence_set_t::status_trigger (C++ member), 455

esp_ble_mesh_sensor_cadence_set_t::status_trigger_property_id (C++ member), 455

esp_ble_mesh_sensor_cadence_status_cb_t (C++ class), 457

esp_ble_mesh_sensor_cadence_status_cb_t::property_id (C++ member), 457

esp_ble_mesh_sensor_cadence_status_cb_t::sensor_cadence (C++ member), 457

esp_ble_mesh_sensor_cadence_t (C++ class), 459

esp_ble_mesh_sensor_cadence_t::fast_cadence_high (C++ member), 459

esp_ble_mesh_sensor_cadence_t::fast_cadence_low (C++ member), 459

esp_ble_mesh_sensor_cadence_t::min_interval (C++ member), 459

esp_ble_mesh_sensor_cadence_t::period_descriptor esp_ble_mesh_sensor_client_status_cb_t::column_status
 (C++ member), 459 (C++ member), 454
 esp_ble_mesh_sensor_cadence_t::trigger_event esp_ble_mesh_sensor_client_status_cb_t::descriptor
 (C++ member), 459 (C++ member), 453
 esp_ble_mesh_sensor_cadence_t::trigger_event_type esp_ble_mesh_sensor_client_status_cb_t::sensor_status
 (C++ member), 459 (C++ member), 454
 esp_ble_mesh_sensor_client_cb_event_t esp_ble_mesh_sensor_client_status_cb_t::setting_status
 (C++ enum), 466 (C++ member), 453
 esp_ble_mesh_sensor_client_cb_param_t esp_ble_mesh_sensor_client_status_cb_t::settings_status
 (C++ class), 458 (C++ member), 453
 esp_ble_mesh_sensor_client_cb_param_t::ESP_BLE_MESH_SENSOR_CLIENT_TIMEOUT_EVT
 (C++ member), 458 (C++ enumerator), 466
 esp_ble_mesh_sensor_client_cb_param_t::ESP_BLE_MESH_SENSOR_COLUMN_GET_T (C++
 (C++ member), 458 class), 456
 esp_ble_mesh_sensor_client_cb_param_t::ESP_BLE_MESH_SENSOR_COLUMN_GET_T::property_id
 (C++ member), 458 (C++ member), 456
 esp_ble_mesh_sensor_client_cb_t (C++ esp_ble_mesh_sensor_column_get_t::raw_value_x
 type), 466 (C++ member), 456
 ESP_BLE_MESH_SENSOR_CLIENT_EVT_MAX esp_ble_mesh_sensor_column_status_cb_t
 (C++ enumerator), 466 (C++ class), 458
 esp_ble_mesh_sensor_client_get_state esp_ble_mesh_sensor_column_status_cb_t::property_status
 (C++ function), 452 (C++ member), 458
 ESP_BLE_MESH_SENSOR_CLIENT_GET_STATE_EVT esp_ble_mesh_sensor_column_status_cb_t::sensor_column_status
 (C++ enumerator), 466 (C++ member), 458
 esp_ble_mesh_sensor_client_get_state_t ESP_BLE_MESH_SENSOR_DATA_FORMAT_A (C
 (C++ union), 453 macro), 465
 esp_ble_mesh_sensor_client_get_state_t::ESP_BLE_MESH_SENSOR_DATA_FORMAT_A_MPID
 (C++ member), 453 (C macro), 465
 esp_ble_mesh_sensor_client_get_state_t::ESP_BLE_MESH_SENSOR_DATA_FORMAT_A_MPID_LEN
 (C++ member), 453 (C macro), 465
 esp_ble_mesh_sensor_client_get_state_t::ESP_BLE_MESH_SENSOR_DATA_FORMAT_B (C
 (C++ member), 453 macro), 465
 esp_ble_mesh_sensor_client_get_state_t::ESP_BLE_MESH_SENSOR_DATA_FORMAT_B_MPID
 (C++ member), 453 (C macro), 466
 esp_ble_mesh_sensor_client_get_state_t::ESP_BLE_MESH_SENSOR_DATA_FORMAT_B_MPID_LEN
 (C++ member), 453 (C macro), 465
 esp_ble_mesh_sensor_client_get_state_t::esp_ble_mesh_sensor_data_t (C++ class),
 (C++ member), 453 459
 esp_ble_mesh_sensor_client_get_state_t::esp_ble_mesh_sensor_data_t::format
 (C++ member), 453 (C++ member), 459
 ESP_BLE_MESH_SENSOR_CLIENT_PUBLISH_EVT esp_ble_mesh_sensor_data_t::length
 (C++ enumerator), 466 (C++ member), 460
 esp_ble_mesh_sensor_client_set_state esp_ble_mesh_sensor_data_t::raw_value
 (C++ function), 452 (C++ member), 460
 ESP_BLE_MESH_SENSOR_CLIENT_SET_STATE_EVT ESP_BLE_MESH_SENSOR_DATA_ZERO_LEN (C
 (C++ enumerator), 466 macro), 465
 esp_ble_mesh_sensor_client_set_state_t esp_ble_mesh_sensor_descriptor_get_t
 (C++ union), 453 (C++ class), 455
 esp_ble_mesh_sensor_client_set_state_t::esp_ble_mesh_sensor_descriptor_get_t::open
 (C++ member), 453 (C++ member), 455
 esp_ble_mesh_sensor_client_set_state_t::esp_ble_mesh_sensor_descriptor_get_t::property_id
 (C++ member), 453 (C++ member), 455
 esp_ble_mesh_sensor_client_status_cb_t ESP_BLE_MESH_SENSOR_DESCRIPTOR_LEN (C
 (C++ union), 453 macro), 464
 esp_ble_mesh_sensor_client_status_cb_t::esp_ble_mesh_sensor_descriptor_status_cb_t
 (C++ member), 453 (C++ class), 457

esp_ble_mesh_sensor_descriptor_status_cb_t::data (C++ union), 467
 (C++ member), 457
 esp_ble_mesh_sensor_descriptor_t (C++ class), 458
 esp_ble_mesh_sensor_descriptor_t::measure_per (C++ member), 463
 (C++ member), 459
 esp_ble_mesh_sensor_descriptor_t::negative_to (C++ member), 463
 (C++ member), 458
 esp_ble_mesh_sensor_descriptor_t::positive_to (C++ member), 463
 (C++ member), 458
 esp_ble_mesh_sensor_descriptor_t::sampling_func (C++ member), 466
 (C++ member), 459
 esp_ble_mesh_sensor_descriptor_t::update_interval (C++ union), 454
 (C++ member), 459
 ESP_BLE_MESH_SENSOR_DIVISOR_TRIGGER_TYPE_LEN (C++ member), 455
 (C macro), 465
 esp_ble_mesh_sensor_get_t (C++ class), 456
 esp_ble_mesh_sensor_get_t::op_en (C++ member), 456
 esp_ble_mesh_sensor_get_t::property_id (C++ member), 456
 esp_ble_mesh_sensor_message_opcode_t (C++ type), 323
 ESP_BLE_MESH_SENSOR_NOT_APPL_MEASURE_PERIOD (C macro), 464
 ESP_BLE_MESH_SENSOR_NOT_APPL_UPDATE_INTERVAL (C macro), 464
 ESP_BLE_MESH_SENSOR_PERIOD_DIVISOR_MAX_VALUE (C macro), 465
 ESP_BLE_MESH_SENSOR_PROPERTY_ID_LEN (C macro), 464
 esp_ble_mesh_sensor_sample_func (C++ enum), 467
 esp_ble_mesh_sensor_series_column_t (C++ class), 460
 esp_ble_mesh_sensor_series_column_t::column_id (C++ member), 460
 esp_ble_mesh_sensor_series_column_t::raw_value (C++ member), 460
 esp_ble_mesh_sensor_series_column_t::raw_value_2 (C++ member), 460
 esp_ble_mesh_sensor_series_get_t (C++ class), 456
 esp_ble_mesh_sensor_series_get_t::op_en (C++ member), 457
 esp_ble_mesh_sensor_series_get_t::property_id (C++ member), 457
 esp_ble_mesh_sensor_series_get_t::raw_value (C++ member), 457
 esp_ble_mesh_sensor_series_get_t::raw_value_2 (C++ member), 457
 esp_ble_mesh_sensor_series_status_cb_t (C++ class), 458
 esp_ble_mesh_sensor_series_status_cb_t::property_id (C++ member), 458
 esp_ble_mesh_sensor_series_status_cb_t::raw_value (C++ member), 458
 esp_ble_mesh_sensor_server_cb_event_t (C++ union), 454
 esp_ble_mesh_sensor_server_cb_param_t (C++ class), 463
 esp_ble_mesh_sensor_server_cb_param_t::ctx (C++ member), 463
 esp_ble_mesh_sensor_server_cb_param_t::model (C++ member), 463
 esp_ble_mesh_sensor_server_cb_param_t::value (C++ member), 463
 esp_ble_mesh_sensor_server_cb_t (C++ union), 454
 esp_ble_mesh_sensor_server_cb_value_t (C++ union), 454
 esp_ble_mesh_sensor_server_cb_value_t::get (C++ member), 455
 esp_ble_mesh_sensor_server_cb_value_t::set (C++ member), 455
 esp_ble_mesh_sensor_server_cb_value_t::state_change (C++ member), 455
 ESP_BLE_MESH_SENSOR_SERVER_EVT_MAX (C++ enumerator), 467
 ESP_BLE_MESH_SENSOR_SERVER_RECV_GET_MSG_EVT (C++ enumerator), 467
 esp_ble_mesh_sensor_server_recv_get_msg_t (C++ union), 454
 esp_ble_mesh_sensor_server_recv_get_msg_t::sensor_id (C++ member), 454
 esp_ble_mesh_sensor_server_recv_get_msg_t::sensor_property_id (C++ member), 454
 esp_ble_mesh_sensor_server_recv_get_msg_t::sensor_value (C++ member), 454
 ESP_BLE_MESH_SENSOR_SERVER_RECV_SET_MSG_EVT (C++ enumerator), 467
 esp_ble_mesh_sensor_server_recv_set_msg_t (C++ union), 454
 esp_ble_mesh_sensor_server_recv_set_msg_t::sensor_id (C++ member), 454
 esp_ble_mesh_sensor_server_recv_set_msg_t::sensor_property_id (C++ member), 454
 ESP_BLE_MESH_SENSOR_SERVER_STATE_CHANGE_EVT (C++ enumerator), 467
 esp_ble_mesh_sensor_server_state_change_t (C++ union), 454
 esp_ble_mesh_sensor_server_state_change_t::sensor_id (C++ member), 454
 esp_ble_mesh_sensor_server_state_change_t::sensor_property_id (C++ member), 454
 ESP_BLE_MESH_SENSOR_SETTING_ACCESS_LEN (C macro), 464
 ESP_BLE_MESH_SENSOR_SETTING_ACCESS_READ (C macro), 464

- (*C macro*), 464
- ESP_BLE_MESH_SENSOR_SETTING_ACCESS_READ_WRITE (*C macro*), 464
- esp_ble_mesh_sensor_setting_get_t (*C++ class*), 456
- esp_ble_mesh_sensor_setting_get_t::sensor_property_id (*C++ member*), 456
- esp_ble_mesh_sensor_setting_get_t::sensor_set_id (*C++ member*), 456
- ESP_BLE_MESH_SENSOR_SETTING_PROPERTY_ID_LEN (*C macro*), 464
- esp_ble_mesh_sensor_setting_set_t (*C++ class*), 456
- esp_ble_mesh_sensor_setting_set_t::sensor_property_id (*C++ member*), 456
- esp_ble_mesh_sensor_setting_set_t::sensor_set_id (*C++ member*), 456
- esp_ble_mesh_sensor_setting_set_t::sensor_set (*C++ member*), 460
- esp_ble_mesh_sensor_setting_status_cb_t (*C++ class*), 457
- esp_ble_mesh_sensor_setting_status_cb_t::open (*C++ member*), 457
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 457
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 457
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- esp_ble_mesh_sensor_setting_status_cb_t::sense (*C++ member*), 458
- ESP_BLE_MESH_SENSOR_STATUS_MIN_INTERVAL_LEN (*C macro*), 465
- ESP_BLE_MESH_SENSOR_STATUS_MIN_INTERVAL_MAX (*C macro*), 465
- ESP_BLE_MESH_SENSOR_STATUS_TRIGGER_TYPE_CHAR (*C macro*), 465
- ESP_BLE_MESH_SENSOR_STATUS_TRIGGER_TYPE_UINT16 (*C macro*), 465
- ESP_BLE_MESH_SENSOR_UNSPECIFIED_NEG_TOLERANCE (*C macro*), 464
- ESP_BLE_MESH_SENSOR_UNSPECIFIED_POS_TOLERANCE (*C macro*), 464
- ESP_BLE_MESH_SERVER_AUTO_RSP (*C macro*), 322
- ESP_BLE_MESH_SERVER_FLAG_MAX (*C++ enumerator*), 329
- esp_ble_mesh_server_model_send_msg (*C++ function*), 335
- ESP_BLE_MESH_SERVER_MODEL_STATE_MAX (*C++ enumerator*), 330
- esp_ble_mesh_server_model_update_state (*C++ function*), 335
- ESP_BLE_MESH_SERVER_MODEL_UPDATE_STATE_COMP_EVT (*C++ enumerator*), 330
- esp_ble_mesh_server_recv_gen_admin_property_get_t (*C++ class*), 392
- esp_ble_mesh_server_recv_gen_admin_property_get_t::sense (*C++ member*), 392

esp_ble_mesh_server_rcv_scene_store_t : esp_ble_mesh_server_rcv_sensor_series_get_t
 (C++ member), 483 (C++ class), 463
 esp_ble_mesh_server_rcv_scheduler_act : esp_ble_mesh_server_rcv_sensor_series_get_t::op
 (C++ class), 482 (C++ member), 463
 esp_ble_mesh_server_rcv_scheduler_act : esp_ble_mesh_server_rcv_sensor_series_get_t::pro
 (C++ member), 482 (C++ member), 463
 esp_ble_mesh_server_rcv_scheduler_act : esp_ble_mesh_server_rcv_sensor_series_get_t::raw
 (C++ class), 483 (C++ member), 463
 esp_ble_mesh_server_rcv_scheduler_act : esp_ble_mesh_server_rcv_sensor_setting_get_t
 (C++ member), 484 (C++ class), 462
 esp_ble_mesh_server_rcv_scheduler_act : esp_ble_mesh_server_rcv_sensor_setting_get_t::pr
 (C++ member), 484 (C++ member), 462
 esp_ble_mesh_server_rcv_scheduler_act : esp_ble_mesh_server_rcv_sensor_setting_get_t::se
 (C++ member), 484 (C++ member), 462
 esp_ble_mesh_server_rcv_scheduler_act : esp_ble_mesh_server_rcv_sensor_setting_set_t
 (C++ member), 484 (C++ class), 463
 esp_ble_mesh_server_rcv_scheduler_act : esp_ble_mesh_server_rcv_sensor_setting_set_t::pr
 (C++ member), 483 (C++ member), 463
 esp_ble_mesh_server_rcv_scheduler_act : esp_ble_mesh_server_rcv_sensor_setting_set_t::se
 (C++ member), 484 (C++ member), 463
 esp_ble_mesh_server_rcv_scheduler_act : esp_ble_mesh_server_rcv_sensor_setting_set_t::se
 (C++ member), 483 (C++ member), 463
 esp_ble_mesh_server_rcv_scheduler_act : esp_ble_mesh_server_rcv_sensor_settings_get_t
 (C++ member), 484 (C++ class), 462
 esp_ble_mesh_server_rcv_scheduler_act : esp_ble_mesh_server_rcv_sensor_settings_get_t::p
 (C++ member), 484 (C++ member), 462
 esp_ble_mesh_server_rcv_scheduler_act : esp_ble_mesh_server_rcv_sensor_status_t
 (C++ member), 484 (C++ class), 447
 esp_ble_mesh_server_rcv_scheduler_act : esp_ble_mesh_server_rcv_sensor_status_t::data
 (C++ member), 483 (C++ member), 447
 esp_ble_mesh_server_rcv_sensor_cadence : esp_ble_mesh_server_rcv_tai_utc_delta_set_t
 (C++ class), 462 (C++ class), 482
 esp_ble_mesh_server_rcv_sensor_cadence : esp_ble_mesh_server_rcv_tai_utc_delta_set_t::pad
 (C++ member), 462 (C++ member), 482
 esp_ble_mesh_server_rcv_sensor_cadence : esp_ble_mesh_server_rcv_tai_utc_delta_set_t::tai
 (C++ class), 463 (C++ member), 483
 esp_ble_mesh_server_rcv_sensor_cadence : esp_ble_mesh_server_rcv_tai_utc_delta_set_t::tai
 (C++ member), 463 (C++ member), 482
 esp_ble_mesh_server_rcv_sensor_cadence : esp_ble_mesh_server_rcv_time_role_set_t
 (C++ member), 463 (C++ class), 483
 esp_ble_mesh_server_rcv_sensor_column : esp_ble_mesh_server_rcv_time_role_set_t::time_ro
 (C++ class), 462 (C++ member), 483
 esp_ble_mesh_server_rcv_sensor_column : esp_ble_mesh_server_rcv_time_set_t
 (C++ member), 463 (C++ class), 482
 esp_ble_mesh_server_rcv_sensor_column : esp_ble_mesh_server_rcv_time_set_t::subsecond
 (C++ member), 463 (C++ member), 482
 esp_ble_mesh_server_rcv_sensor_descriptors : esp_ble_mesh_server_rcv_time_set_t::tai_seconds
 (C++ class), 462 (C++ member), 482
 esp_ble_mesh_server_rcv_sensor_descriptors : esp_ble_mesh_server_rcv_time_set_t::tai_utc_delt
 (C++ member), 462 (C++ member), 482
 esp_ble_mesh_server_rcv_sensor_descriptors : esp_ble_mesh_server_rcv_time_set_t::time_authori
 (C++ member), 462 (C++ member), 482
 esp_ble_mesh_server_rcv_sensor_get_t : esp_ble_mesh_server_rcv_time_set_t::time_zone_of
 (C++ class), 462 (C++ member), 482
 esp_ble_mesh_server_rcv_sensor_get_t : esp_ble_mesh_server_rcv_time_set_t::uncertainty
 (C++ member), 462 (C++ member), 482
 esp_ble_mesh_server_rcv_sensor_get_t : esp_ble_mesh_server_rcv_time_status_t
 (C++ member), 462 (C++ class), 484

esp_ble_mesh_server_rcv_time_status_t::esp_ble_mesh_server_state_value_t::light_lc_light
 (C++ member), 484 (C++ member), 298
 esp_ble_mesh_server_rcv_time_status_t::esp_ble_mesh_server_state_value_t::light_lightnes
 (C++ member), 484 (C++ member), 297
 esp_ble_mesh_server_rcv_time_status_t::esp_ble_mesh_server_state_value_t::light_lightnes
 (C++ member), 484 (C++ member), 297
 esp_ble_mesh_server_rcv_time_status_t::esp_ble_mesh_server_state_value_t::light_xyl_ligh
 (C++ member), 484 (C++ member), 298
 esp_ble_mesh_server_rcv_time_status_t::esp_ble_mesh_server_state_value_t::lightness
 (C++ member), 484 (C++ member), 297
 esp_ble_mesh_server_rcv_time_status_t::esp_ble_mesh_server_state_value_t::onoff
 (C++ member), 484 (C++ member), 297
 esp_ble_mesh_server_rcv_time_zone_set_esp_ble_mesh_server_state_value_t::onpowerup
 (C++ class), 482 (C++ member), 297
 esp_ble_mesh_server_rcv_time_zone_set_esp_ble_mesh_server_state_value_t::power
 (C++ member), 482 (C++ member), 297
 esp_ble_mesh_server_rcv_time_zone_set_esp_ble_mesh_server_state_value_t::saturation
 (C++ member), 482 (C++ member), 298
 ESP_BLE_MESH_SERVER_RSP_BY_APP (C esp_ble_mesh_server_state_value_t::temperature
 macro), 322 (C++ member), 297
 esp_ble_mesh_server_rsp_ctrl_t (C++ ESP_BLE_MESH_SERVER_TRANS_TIMER_START
 class), 307 (C++ enumerator), 329
 esp_ble_mesh_server_rsp_ctrl_t::get_auto_esp_ble_mesh_set_fast_prov_action
 (C++ member), 308 (C++ function), 345
 esp_ble_mesh_server_rsp_ctrl_t::set_auto_esp_ble_mesh_set_fast_prov_action_comp_evt
 (C++ member), 308 (C++ enumerator), 328
 esp_ble_mesh_server_rsp_ctrl_t::status_esp_ble_mesh_set_fast_prov_info (C++
 (C++ member), 308 function), 345
 esp_ble_mesh_server_state_type_t (C++ ESP_BLE_MESH_SET_FAST_PROV_INFO_COMP_EVT
 enum), 329 (C++ enumerator), 328
 esp_ble_mesh_server_state_value_t esp_ble_mesh_set_unprovisioned_device_name
 (C++ union), 297 (C++ function), 342
 esp_ble_mesh_server_state_value_t::delta_esp_ble_mesh_settings_uid_size (C
 (C++ member), 297 macro), 308
 esp_ble_mesh_server_state_value_t::gen_esp_ble_mesh_sig_model (C macro), 310
 (C++ member), 297 esp_ble_mesh_state_change_cfg_appkey_add_t
 esp_ble_mesh_server_state_value_t::gen_onoff (C++ class), 368
 (C++ member), 297 esp_ble_mesh_state_change_cfg_appkey_add_t::app_i
 (C++ member), 368
 esp_ble_mesh_server_state_value_t::gen_onpowerup (C++ member), 368
 (C++ member), 297 esp_ble_mesh_state_change_cfg_appkey_add_t::app_k
 (C++ member), 368
 esp_ble_mesh_server_state_value_t::gen_power_level (C++ member), 368
 (C++ member), 297 esp_ble_mesh_state_change_cfg_appkey_add_t::net_i
 (C++ member), 368
 esp_ble_mesh_server_state_value_t::hue (C++ member), 368
 (C++ member), 298 esp_ble_mesh_state_change_cfg_appkey_delete_t
 esp_ble_mesh_server_state_value_t::level (C++ class), 369
 (C++ member), 297 esp_ble_mesh_state_change_cfg_appkey_delete_t::ap
 (C++ member), 369
 esp_ble_mesh_server_state_value_t::light_ctl (C++ member), 369
 (C++ member), 297 esp_ble_mesh_state_change_cfg_appkey_delete_t::ne
 (C++ member), 298 esp_ble_mesh_state_change_cfg_appkey_update_t
 (C++ member), 368
 esp_ble_mesh_server_state_value_t::light_hsl (C++ class), 368
 (C++ member), 298 esp_ble_mesh_state_change_cfg_appkey_update_t::ap
 (C++ member), 369
 esp_ble_mesh_server_state_value_t::light_hsl (C++ member), 369
 (C++ member), 298 esp_ble_mesh_state_change_cfg_appkey_update_t::ap
 (C++ member), 369
 esp_ble_mesh_server_state_value_t::light_hsl (C++ member), 369
 (C++ member), 298 esp_ble_mesh_state_change_cfg_appkey_update_t::ne
 (C++ member), 369
 esp_ble_mesh_server_state_value_t::light_hsl (C++ member), 369
 (C++ member), 298 esp_ble_mesh_state_change_cfg_kr_phase_set_t

(C++ member), 441

esp_ble_mesh_state_change_sensor_status_set_t:ble_mesh_state_transition_t::delay (C++ member), 441

esp_ble_mesh_state_change_sensor_status_set_t:ble_mesh_state_transition_t::just_started (C++ member), 441

esp_ble_mesh_state_change_sensor_status_set_t:ble_mesh_state_transition_t::quod_tt (C++ member), 441

esp_ble_mesh_state_change_sensor_status_set_t:ble_mesh_state_transition_t::remain_time (C++ member), 441

esp_ble_mesh_state_change_tai_utc_delta_set_t:ble_mesh_state_transition_t::start_timestamp (C++ class), 480

esp_ble_mesh_state_change_tai_utc_delta_set_t:ble_mesh_state_transition_t::timer (C++ member), 480

esp_ble_mesh_state_change_tai_utc_delta_set_t:ble_mesh_state_transition_t::total_duration (C++ member), 480

esp_ble_mesh_state_change_time_role_set_t:ble_mesh_state_transition_t::trans_time (C++ class), 481

esp_ble_mesh_state_change_time_role_set_t:ESP_BLE_MESH_STATIC_OOB (C++ enumerator), (C++ member), 481

esp_ble_mesh_state_change_time_set_t ESP_BLE_MESH_SUPPLY_VOLTAGE_TOO_HIGH_ERROR (C++ class), 479

esp_ble_mesh_state_change_time_set_t::ESP_BLE_MESH_SUPPLY_VOLTAGE_TOO_HIGH_WARNING (C++ member), 480

esp_ble_mesh_state_change_time_set_t::ESP_BLE_MESH_SUPPLY_VOLTAGE_TOO_LOW_ERROR (C++ member), 480

esp_ble_mesh_state_change_time_set_t::ESP_BLE_MESH_SUPPLY_VOLTAGE_TOO_LOW_WARNING (C++ member), 480

esp_ble_mesh_state_change_time_set_t::ESP_BLE_MESH_TAI_OF_DELTA_CHANGE_LEN (C++ member), 480

esp_ble_mesh_state_change_time_set_t::ESP_BLE_MESH_TAI_OF_ZONE_CHANGE_LEN (C++ member), 480

esp_ble_mesh_state_change_time_set_t::ESP_BLE_MESH_TAI_SECONDS_LEN (C++ member), 480

esp_ble_mesh_state_change_time_status_tESP_BLE_MESH_TAI_UTC_DELTA_MAX_VALUE (C++ class), 480

esp_ble_mesh_state_change_time_status_t:esp_ble_mesh_tai_utc_delta_set_t (C++ member), 480

esp_ble_mesh_state_change_time_status_t:esp_ble_mesh_tai_utc_delta_set_t::padding (C++ member), 480

esp_ble_mesh_state_change_time_status_t:esp_ble_mesh_tai_utc_delta_set_t::tai_delta_change (C++ member), 480

esp_ble_mesh_state_change_time_status_t:esp_ble_mesh_tai_utc_delta_set_t::tai_utc_delta (C++ member), 480

esp_ble_mesh_state_change_time_status_t:esp_ble_mesh_offset_delta_status_cb_t (C++ member), 480

esp_ble_mesh_state_change_time_status_t:esp_ble_mesh_tai_utc_delta_status_cb_t::padding_1 (C++ member), 480

esp_ble_mesh_state_change_time_status_t:esp_ble_mesh_tai_utc_delta_status_cb_t::padding_2 (C++ class), 480

esp_ble_mesh_state_change_time_status_t:esp_ble_mesh_tai_utc_delta_status_cb_t::tai_delta_change (C++ member), 480

esp_ble_mesh_state_change_time_status_t:esp_ble_mesh_tai_utc_delta_status_cb_t::tai_utc_delta (C++ member), 480

esp_ble_mesh_state_transition_t (C++ esp_ble_mesh_tai_utc_delta_status_cb_t::tai_utc_delta class), 307

esp_ble_mesh_state_transition_t::BLE_MESH_STATIC_OOB (C++ member), 474

esp_ble_mesh_state_transition_t::BLE_MESH_TAMPER_ERROR (C macro), 409

esp_ble_mesh_state_transition_t::BLE_MESH_TAMPER_WARNING (C macro), 409

esp_ble_mesh_state_transition_t::counter 409

- esp_ble_mesh_time_zone_status_cb_t (C++ class), 473
 esp_ble_mesh_time_zone_status_cb_t::tai_zone_change (C++ member), 473
 esp_ble_mesh_time_zone_status_cb_t::time_zone_curr (C++ member), 473
 esp_ble_mesh_time_zone_status_cb_t::time_zone_offset_new (C++ member), 473
 ESP_BLE_MESH_TRANSMIT (C macro), 309
 ESP_BLE_MESH_TTL_DEFAULT (C macro), 308
 ESP_BLE_MESH_TTL_MAX (C macro), 308
 ESP_BLE_MESH_TWIST (C++ enumerator), 325
 ESP_BLE_MESH_TYPE_COMPLETE_CB (C++ enumerator), 324
 ESP_BLE_MESH_TYPE_INTPUT_CB (C++ enumerator), 324
 ESP_BLE_MESH_TYPE_LINK_CLOSE_CB (C++ enumerator), 324
 ESP_BLE_MESH_TYPE_LINK_OPEN_CB (C++ enumerator), 324
 ESP_BLE_MESH_TYPE_OUTPUT_NUM_CB (C++ enumerator), 324
 ESP_BLE_MESH_TYPE_OUTPUT_STR_CB (C++ enumerator), 324
 ESP_BLE_MESH_TYPE_PROV_CB (C++ enumerator), 324
 ESP_BLE_MESH_TYPE_RESET_CB (C++ enumerator), 324
 ESP_BLE_MESH_UNKNOWN_TAI_DELTA_CHANGE (C macro), 486
 ESP_BLE_MESH_UNKNOWN_TAI_SECONDS (C macro), 486
 ESP_BLE_MESH_UNKNOWN_TAI_ZONE_CHANGE (C macro), 486
 esp_ble_mesh_unprov_dev_add_t (C++ class), 303
 esp_ble_mesh_unprov_dev_add_t::addr (C++ member), 304
 esp_ble_mesh_unprov_dev_add_t::addr_type (C++ member), 304
 esp_ble_mesh_unprov_dev_add_t::bearer (C++ member), 304
 esp_ble_mesh_unprov_dev_add_t::oob_info (C++ member), 304
 esp_ble_mesh_unprov_dev_add_t::uuid (C++ member), 304
 ESP_BLE_MESH_VENDOR_MODEL (C macro), 310
 ESP_BLE_MESH_VIBRATE (C++ enumerator), 324
 ESP_BLE_MESH_VIBRATION_ERROR (C macro), 408
 ESP_BLE_MESH_VIBRATION_WARNING (C macro), 408
 ESP_BLE_ONLY_ACCEPT_SPECIFIED_AUTH_DISABLE (C macro), 143
 ESP_BLE_ONLY_ACCEPT_SPECIFIED_AUTH_ENABLE (C macro), 144
 ESP_BLE_OOB_DISABLE (C macro), 144
 ESP_BLE_OOB_ENABLE (C macro), 144
 esp_ble_oob_req_reply (C++ function), 130
 esp_ble_passkey_reply (C++ function), 130
 esp_ble_penc_keys_t (C++ class), 140
 esp_ble_penc_keys_t::counter (C++ member), 140
 esp_ble_penc_keys_t::csrkc (C++ member), 140
 esp_ble_penc_keys_t::sec_level (C++ member), 140
 esp_ble_penc_keys_t (C++ class), 140
 esp_ble_penc_keys_t::ediv (C++ member), 140
 esp_ble_penc_keys_t::key_size (C++ member), 140
 esp_ble_penc_keys_t::ltk (C++ member), 140
 esp_ble_penc_keys_t::rand (C++ member), 140
 esp_ble_penc_keys_t::sec_level (C++ member), 140
 esp_ble_pid_keys_t (C++ class), 140
 esp_ble_pid_keys_t::addr_type (C++ member), 140
 esp_ble_pid_keys_t::irk (C++ member), 140
 esp_ble_pid_keys_t::static_addr (C++ member), 140
 esp_ble_pkt_data_length_params_t (C++ class), 140
 esp_ble_pkt_data_length_params_t::rx_len (C++ member), 140
 esp_ble_pkt_data_length_params_t::tx_len (C++ member), 140
 esp_ble_power_type_t (C++ enum), 119
 ESP_BLE_PWR_TYPE_ADV (C++ enumerator), 119
 ESP_BLE_PWR_TYPE_CONN_HDL0 (C++ enumerator), 119
 ESP_BLE_PWR_TYPE_CONN_HDL1 (C++ enumerator), 119
 ESP_BLE_PWR_TYPE_CONN_HDL2 (C++ enumerator), 119
 ESP_BLE_PWR_TYPE_CONN_HDL3 (C++ enumerator), 119
 ESP_BLE_PWR_TYPE_CONN_HDL4 (C++ enumerator), 119
 ESP_BLE_PWR_TYPE_CONN_HDL5 (C++ enumerator), 119
 ESP_BLE_PWR_TYPE_CONN_HDL6 (C++ enumerator), 119
 ESP_BLE_PWR_TYPE_CONN_HDL7 (C++ enumerator), 119
 ESP_BLE_PWR_TYPE_CONN_HDL8 (C++ enumerator), 119
 ESP_BLE_PWR_TYPE_DEFAULT (C++ enumerator), 119
 ESP_BLE_PWR_TYPE_NUM (C++ enumerator), 119
 ESP_BLE_PWR_TYPE_SCAN (C++ enumerator), 119
 esp_ble_remove_bond_device (C++ function), 130

- esp_ble_resolve_adv_data (C++ function), 128
- ESP_BLE_SCA_100PPM (C++ enumerator), 118
- ESP_BLE_SCA_150PPM (C++ enumerator), 118
- ESP_BLE_SCA_20PPM (C++ enumerator), 118
- ESP_BLE_SCA_250PPM (C++ enumerator), 118
- ESP_BLE_SCA_30PPM (C++ enumerator), 118
- ESP_BLE_SCA_500PPM (C++ enumerator), 118
- ESP_BLE_SCA_50PPM (C++ enumerator), 118
- ESP_BLE_SCA_75PPM (C++ enumerator), 118
- esp_ble_scan_duplicate_list_flush (C++ function), 116
- esp_ble_scan_duplicate_t (C++ enum), 150
- esp_ble_scan_filter_t (C++ enum), 149
- ESP_BLE_SCAN_PARAM_UNDEF (C macro), 121
- esp_ble_scan_params_t (C++ class), 138
- esp_ble_scan_params_t::own_addr_type (C++ member), 139
- esp_ble_scan_params_t::scan_duplicate (C++ member), 139
- esp_ble_scan_params_t::scan_filter_policy (C++ member), 139
- esp_ble_scan_params_t::scan_interval (C++ member), 139
- esp_ble_scan_params_t::scan_type (C++ member), 139
- esp_ble_scan_params_t::scan_window (C++ member), 139
- ESP_BLE_SCAN_RSP_DATA_LEN_MAX (C macro), 145
- esp_ble_scan_type_t (C++ enum), 149
- esp_ble_sec_act_t (C++ enum), 149
- ESP_BLE_SEC_ENCRYPT (C++ enumerator), 149
- ESP_BLE_SEC_ENCRYPT_MITM (C++ enumerator), 149
- ESP_BLE_SEC_ENCRYPT_NO_MITM (C++ enumerator), 149
- esp_ble_sec_key_notif_t (C++ class), 141
- esp_ble_sec_key_notif_t::bd_addr (C++ member), 141
- esp_ble_sec_key_notif_t::passkey (C++ member), 141
- esp_ble_sec_req_t (C++ class), 141
- esp_ble_sec_req_t::bd_addr (C++ member), 141
- esp_ble_sec_t (C++ union), 131
- esp_ble_sec_t::auth_cmpl (C++ member), 132
- esp_ble_sec_t::ble_id_keys (C++ member), 132
- esp_ble_sec_t::ble_key (C++ member), 132
- esp_ble_sec_t::ble_req (C++ member), 132
- esp_ble_sec_t::key_notif (C++ member), 132
- esp_ble_set_encryption (C++ function), 129
- ESP_BLE_SM_AUTHEN_REQ_MODE (C++ enumerator), 149
- ESP_BLE_SM_CLEAR_STATIC_PASSKEY (C++ enumerator), 149
- ESP_BLE_SM_IOCTL_CAP_MODE (C++ enumerator), 149
- ESP_BLE_SM_MAX_KEY_SIZE (C++ enumerator), 149
- ESP_BLE_SM_MAX_PARAM (C++ enumerator), 149
- ESP_BLE_SM_MIN_KEY_SIZE (C++ enumerator), 149
- ESP_BLE_SM_ONLY_ACCEPT_SPECIFIED_SEC_AUTH (C++ enumerator), 149
- ESP_BLE_SM_OOB_SUPPORT (C++ enumerator), 149
- esp_ble_sm_param_t (C++ enum), 149
- ESP_BLE_SM_PASSKEY (C++ enumerator), 149
- ESP_BLE_SM_SET_INIT_KEY (C++ enumerator), 149
- ESP_BLE_SM_SET_RSP_KEY (C++ enumerator), 149
- ESP_BLE_SM_SET_STATIC_PASSKEY (C++ enumerator), 149
- esp_ble_tx_power_get (C++ function), 113
- esp_ble_tx_power_set (C++ function), 113
- ESP_BLE_WHITELIST_ADD (C++ enumerator), 151
- ESP_BLE_WHITELIST_REMOVE (C++ enumerator), 151
- esp_ble_wl_addr_type_t (C++ enum), 123
- esp_ble_wl_operation_t (C++ enum), 151
- esp_blueandroid_deinit (C++ function), 123
- esp_blueandroid_disable (C++ function), 123
- esp_blueandroid_enable (C++ function), 123
- esp_blueandroid_get_status (C++ function), 123
- esp_blueandroid_init (C++ function), 123
- ESP_BLUEANDROID_STATUS_CHECK (C macro), 121
- ESP_BLUEANDROID_STATUS_ENABLED (C++ enumerator), 124
- ESP_BLUEANDROID_STATUS_INITIALIZED (C++ enumerator), 124
- esp_blueandroid_status_t (C++ enum), 124
- ESP_BLUEANDROID_STATUS_UNINITIALIZED (C++ enumerator), 124
- esp_blufi_ap_record_t (C++ class), 197
- esp_blufi_ap_record_t::rssi (C++ member), 198
- esp_blufi_ap_record_t::ssid (C++ member), 198
- esp_blufi_callbacks_t (C++ class), 198
- esp_blufi_callbacks_t::checksum_func (C++ member), 198
- esp_blufi_callbacks_t::decrypt_func (C++ member), 198
- esp_blufi_callbacks_t::encrypt_func (C++ member), 198
- esp_blufi_callbacks_t::event_cb (C++ member), 198
- esp_blufi_callbacks_t::negotiate_data_handler (C++ member), 198
- esp_blufi_cb_event_t (C++ enum), 199
- esp_blufi_cb_param_t (C++ union), 192

- esp_blufi_cb_param_t::connect (C++ member), 192
 esp_blufi_cb_param_t::custom_data (C++ member), 193
 esp_blufi_cb_param_t::deinit_finish (C++ member), 192
 esp_blufi_cb_param_t::disconnect (C++ member), 192
 esp_blufi_cb_param_t::init_finish (C++ member), 192
 esp_blufi_cb_param_t::report_error (C++ member), 193
 esp_blufi_cb_param_t::server_cert (C++ member), 193
 esp_blufi_cb_param_t::server_pkey (C++ member), 193
 esp_blufi_cb_param_t::softap_auth_mode (C++ member), 192
 esp_blufi_cb_param_t::softap_channel (C++ member), 192
 esp_blufi_cb_param_t::softap_max_conn_num (C++ member), 192
 esp_blufi_cb_param_t::softap_passwd (C++ member), 192
 esp_blufi_cb_param_t::softap_ssid (C++ member), 192
 esp_blufi_cb_param_t::sta_bssid (C++ member), 192
 esp_blufi_cb_param_t::sta_passwd (C++ member), 192
 esp_blufi_cb_param_t::sta_ssid (C++ member), 192
 esp_blufi_cb_param_t::username (C++ member), 192
 esp_blufi_cb_param_t::wifi_mode (C++ member), 192
 ESP_BLUFI_CHECKSUM_ERROR (C++ enumerator), 200
 esp_blufi_checksum_func_t (C++ type), 199
 esp_blufi_close (C++ function), 191
 ESP_BLUFI_DATA_FORMAT_ERROR (C++ enumerator), 200
 ESP_BLUFI_DECRYPT_ERROR (C++ enumerator), 200
 esp_blufi_decrypt_func_t (C++ type), 198
 ESP_BLUFI_DEINIT_FAILED (C++ enumerator), 200
 ESP_BLUFI_DEINIT_OK (C++ enumerator), 200
 esp_blufi_deinit_state_t (C++ enum), 200
 ESP_BLUFI_DH_MALLOC_ERROR (C++ enumerator), 200
 ESP_BLUFI_DH_PARAM_ERROR (C++ enumerator), 200
 ESP_BLUFI_ENCRYPT_ERROR (C++ enumerator), 200
 esp_blufi_encrypt_func_t (C++ type), 198
 esp_blufi_error_state_t (C++ enum), 200
 ESP_BLUFI_EVENT_BLE_CONNECT (C++ enumerator), 199
 ESP_BLUFI_EVENT_BLE_DISCONNECT (C++ enumerator), 199
 esp_blufi_event_cb_t (C++ type), 198
 ESP_BLUFI_EVENT_DEAUTHENTICATE_STA (C++ enumerator), 199
 ESP_BLUFI_EVENT_DEINIT_FINISH (C++ enumerator), 199
 ESP_BLUFI_EVENT_GET_WIFI_LIST (C++ enumerator), 199
 ESP_BLUFI_EVENT_GET_WIFI_STATUS (C++ enumerator), 199
 ESP_BLUFI_EVENT_INIT_FINISH (C++ enumerator), 199
 ESP_BLUFI_EVENT_RECV_CA_CERT (C++ enumerator), 199
 ESP_BLUFI_EVENT_RECV_CLIENT_CERT (C++ enumerator), 199
 ESP_BLUFI_EVENT_RECV_CLIENT_PRIV_KEY (C++ enumerator), 199
 ESP_BLUFI_EVENT_RECV_CUSTOM_DATA (C++ enumerator), 199
 ESP_BLUFI_EVENT_RECV_SERVER_CERT (C++ enumerator), 199
 ESP_BLUFI_EVENT_RECV_SERVER_PRIV_KEY (C++ enumerator), 199
 ESP_BLUFI_EVENT_RECV_SLAVE_DISCONNECT_BLE (C++ enumerator), 199
 ESP_BLUFI_EVENT_RECV_SOFTAP_AUTH_MODE (C++ enumerator), 199
 ESP_BLUFI_EVENT_RECV_SOFTAP_CHANNEL (C++ enumerator), 199
 ESP_BLUFI_EVENT_RECV_SOFTAP_MAX_CONN_NUM (C++ enumerator), 199
 ESP_BLUFI_EVENT_RECV_SOFTAP_PASSWD (C++ enumerator), 199
 ESP_BLUFI_EVENT_RECV_SOFTAP_SSID (C++ enumerator), 199
 ESP_BLUFI_EVENT_RECV_STA_BSSID (C++ enumerator), 199
 ESP_BLUFI_EVENT_RECV_STA_PASSWD (C++ enumerator), 199
 ESP_BLUFI_EVENT_RECV_STA_SSID (C++ enumerator), 199
 ESP_BLUFI_EVENT_RECV_USERNAME (C++ enumerator), 199
 ESP_BLUFI_EVENT_REPORT_ERROR (C++ enumerator), 199
 ESP_BLUFI_EVENT_REQ_CONNECT_TO_AP (C++ enumerator), 199
 ESP_BLUFI_EVENT_REQ_DISCONNECT_FROM_AP (C++ enumerator), 199
 ESP_BLUFI_EVENT_SET_WIFI_OPMODE (C++ enumerator), 199
 esp_blufi_extra_info_t (C++ class), 197
 esp_blufi_extra_info_t::softap_authmode (C++ member), 197
 esp_blufi_extra_info_t::softap_authmode_set

- (C++ member), 197
- esp_blufi_extra_info_t::softap_channel (C++ member), 197
- esp_blufi_extra_info_t::softap_channel_set (C++ member), 197
- esp_blufi_extra_info_t::softap_max_conn_num (C++ member), 197
- esp_blufi_extra_info_t::softap_max_conn_num_set (C++ member), 197
- esp_blufi_extra_info_t::softap_passwd (C++ member), 197
- esp_blufi_extra_info_t::softap_passwd_len (C++ member), 197
- esp_blufi_extra_info_t::softap_ssid (C++ member), 197
- esp_blufi_extra_info_t::softap_ssid_len (C++ member), 197
- esp_blufi_extra_info_t::sta_bssid (C++ member), 197
- esp_blufi_extra_info_t::sta_bssid_set (C++ member), 197
- esp_blufi_extra_info_t::sta_passwd (C++ member), 197
- esp_blufi_extra_info_t::sta_passwd_len (C++ member), 197
- esp_blufi_extra_info_t::sta_ssid (C++ member), 197
- esp_blufi_extra_info_t::sta_ssid_len (C++ member), 197
- esp_blufi_get_version (C++ function), 191
- ESP_BLUFI_INIT_FAILED (C++ enumerator), 200
- ESP_BLUFI_INIT_OK (C++ enumerator), 200
- ESP_BLUFI_INIT_SECURITY_ERROR (C++ enumerator), 200
- esp_blufi_init_state_t (C++ enum), 200
- ESP_BLUFI_MAKE_PUBLIC_ERROR (C++ enumerator), 200
- esp_blufi_negotiate_data_handler_t (C++ type), 198
- esp_blufi_profile_deinit (C++ function), 191
- esp_blufi_profile_init (C++ function), 191
- ESP_BLUFI_READ_PARAM_ERROR (C++ enumerator), 200
- esp_blufi_register_callbacks (C++ function), 191
- esp_blufi_send_custom_data (C++ function), 192
- esp_blufi_send_error_info (C++ function), 192
- esp_blufi_send_wifi_conn_report (C++ function), 191
- esp_blufi_send_wifi_list (C++ function), 191
- ESP_BLUFI_SEQUENCE_ERROR (C++ enumerator), 200
- ESP_BLUFI_STA_CONN_FAIL (C++ enumerator), 200
- esp_blufi_sta_conn_state_t (C++ enum), 200
- ESP_BLUFI_STA_CONN_SUCCESS (C++ enumerator), 200
- esp_bredr_sco_datapath_set (C++ function), 114
- esp_bredr_tx_power_get (C++ function), 114
- esp_bredr_tx_power_set (C++ function), 113
- ESP_BT_CLR_COD_SERVICE_CLASS (C++ enumerator), 212
- ESP_BT_COD_FORMAT_TYPE_1 (C macro), 212
- ESP_BT_COD_FORMAT_TYPE_BIT_MASK (C macro), 211
- ESP_BT_COD_FORMAT_TYPE_BIT_OFFSET (C macro), 211
- ESP_BT_COD_MAJOR_DEV_AV (C++ enumerator), 214
- ESP_BT_COD_MAJOR_DEV_BIT_MASK (C macro), 211
- ESP_BT_COD_MAJOR_DEV_BIT_OFFSET (C macro), 211
- ESP_BT_COD_MAJOR_DEV_COMPUTER (C++ enumerator), 214
- ESP_BT_COD_MAJOR_DEV_HEALTH (C++ enumerator), 214
- ESP_BT_COD_MAJOR_DEV_IMAGING (C++ enumerator), 214
- ESP_BT_COD_MAJOR_DEV_LAN_NAP (C++ enumerator), 214
- ESP_BT_COD_MAJOR_DEV_MISC (C++ enumerator), 214
- ESP_BT_COD_MAJOR_DEV_PERIPHERAL (C++ enumerator), 214
- ESP_BT_COD_MAJOR_DEV_PHONE (C++ enumerator), 214
- esp_bt_cod_major_dev_t (C++ enum), 214
- ESP_BT_COD_MAJOR_DEV_TOY (C++ enumerator), 214
- ESP_BT_COD_MAJOR_DEV_UNCATEGORIZED (C++ enumerator), 214
- ESP_BT_COD_MAJOR_DEV_WEARABLE (C++ enumerator), 214
- ESP_BT_COD_MINOR_DEV_BIT_MASK (C macro), 211
- ESP_BT_COD_MINOR_DEV_BIT_OFFSET (C macro), 211
- esp_bt_cod_mode_t (C++ enum), 212
- ESP_BT_COD_SRVC_AUDIO (C++ enumerator), 213
- ESP_BT_COD_SRVC_BIT_MASK (C macro), 211
- ESP_BT_COD_SRVC_BIT_OFFSET (C macro), 211
- ESP_BT_COD_SRVC_CAPTURING (C++ enumerator), 213
- ESP_BT_COD_SRVC_INFORMATION (C++ enumerator), 213
- ESP_BT_COD_SRVC_LMTD_DISCOVER (C++ enumerator), 213
- ESP_BT_COD_SRVC_NETWORKING (C++ enumerator), 213

- ESP_BT_COD_SRVC_NONE (C++ enumerator), 213
- ESP_BT_COD_SRVC_OBJ_TRANSFER (C++ enumerator), 213
- ESP_BT_COD_SRVC_POSITIONING (C++ enumerator), 213
- ESP_BT_COD_SRVC_RENDERING (C++ enumerator), 213
- esp_bt_cod_srvc_t (C++ enum), 213
- ESP_BT_COD_SRVC_TELEPHONY (C++ enumerator), 213
- esp_bt_cod_t (C++ class), 209
- esp_bt_cod_t::major (C++ member), 209
- esp_bt_cod_t::minor (C++ member), 209
- esp_bt_cod_t::reserved_2 (C++ member), 209
- esp_bt_cod_t::reserved_8 (C++ member), 209
- esp_bt_cod_t::service (C++ member), 209
- ESP_BT_CONNECTABLE (C++ enumerator), 212
- esp_bt_connection_mode_t (C++ enum), 212
- ESP_BT_CONTROLLER_CONFIG_MAGIC_VAL (C macro), 118
- esp_bt_controller_config_t (C++ class), 116
- esp_bt_controller_config_t::auto_latency (C++ member), 117
- esp_bt_controller_config_t::ble_max_conn (C++ member), 117
- esp_bt_controller_config_t::ble_sca (C++ member), 117
- esp_bt_controller_config_t::bt_legacy_auth_vs_ble (C++ member), 117
- esp_bt_controller_config_t::bt_max_acl_entries (C++ member), 117
- esp_bt_controller_config_t::bt_max_sync_connections (C++ member), 117
- esp_bt_controller_config_t::bt_sco_datapath (C++ member), 117
- esp_bt_controller_config_t::controller_debug_flag (C++ member), 117
- esp_bt_controller_config_t::controller_task_priority (C++ member), 116
- esp_bt_controller_config_t::controller_task_stack_size (C++ member), 116
- esp_bt_controller_config_t::hci_uart_baudrate (C++ member), 116
- esp_bt_controller_config_t::hci_uart_no (C++ member), 116
- esp_bt_controller_config_t::hli (C++ member), 117
- esp_bt_controller_config_t::magic (C++ member), 117
- esp_bt_controller_config_t::mesh_adv_size (C++ member), 117
- esp_bt_controller_config_t::mode (C++ member), 117
- esp_bt_controller_config_t::normal_adv_size (C++ member), 117
- esp_bt_controller_config_t::pcm_polar (C++ member), 117
- esp_bt_controller_config_t::pcm_role (C++ member), 117
- esp_bt_controller_config_t::scan_duplicate_mode (C++ member), 116
- esp_bt_controller_config_t::scan_duplicate_type (C++ member), 117
- esp_bt_controller_config_t::send_adv_reserved_size (C++ member), 117
- esp_bt_controller_deinit (C++ function), 114
- esp_bt_controller_disable (C++ function), 114
- esp_bt_controller_enable (C++ function), 114
- esp_bt_controller_get_status (C++ function), 114
- esp_bt_controller_init (C++ function), 114
- esp_bt_controller_mem_release (C++ function), 115
- ESP_BT_CONTROLLER_STATUS_ENABLED (C++ enumerator), 118
- ESP_BT_CONTROLLER_STATUS_IDLE (C++ enumerator), 118
- ESP_BT_CONTROLLER_STATUS_INITED (C++ enumerator), 118
- ESP_BT_CONTROLLER_STATUS_NUM (C++ enumerator), 118
- esp_bt_controller_status_t (C++ enum), 118
- esp_bt_dev_get_address (C++ function), 124
- esp_bt_dev_set_device_name (C++ function), 124
- esp_bt_dev_type_t (C++ enum), 122
- ESP_BT_DEVICE_TYPE_BLE (C++ enumerator), 122
- ESP_BT_DEVICE_TYPE_BREDR (C++ enumerator), 122
- ESP_BT_DEVICE_TYPE_DUMO (C++ enumerator), 122
- esp_bt_discovery_mode_t (C++ enum), 212
- esp_bt_exceptional_subcode_type_t (C++ enum), 151
- esp_bt_eir_data_t (C++ class), 209
- esp_bt_eir_data_t::fec_required (C++ member), 210
- esp_bt_eir_data_t::flag (C++ member), 210
- esp_bt_eir_data_t::include_txpower (C++ member), 210
- esp_bt_eir_data_t::include_uuid (C++ member), 210
- esp_bt_eir_data_t::manufacturer_len (C++ member), 210
- esp_bt_eir_data_t::p_manufacturer_data (C++ member), 210
- esp_bt_eir_data_t::p_url (C++ member), 210

- esp_bt_gap_cb_param_t::qos_cmpl (C++ member), 206
 esp_bt_gap_cb_param_t::qos_cmpl_param (C++ class), 208
 esp_bt_gap_cb_param_t::qos_cmpl_param::bda (C++ member), 208
 esp_bt_gap_cb_param_t::qos_cmpl_param::stat (C++ member), 208
 esp_bt_gap_cb_param_t::qos_cmpl_param::t_poll (C++ member), 208
 esp_bt_gap_cb_param_t::read_rmt_name (C++ member), 206
 esp_bt_gap_cb_param_t::read_rmt_name_param (C++ class), 208
 esp_bt_gap_cb_param_t::read_rmt_name_param (C++ member), 208
 esp_bt_gap_cb_param_t::read_rmt_name_param::bda (C++ member), 208
 esp_bt_gap_cb_param_t::read_rssi_delta (C++ member), 205
 esp_bt_gap_cb_param_t::read_rssi_delta_param (C++ class), 208
 esp_bt_gap_cb_param_t::read_rssi_delta_param (C++ member), 208
 esp_bt_gap_cb_param_t::read_rssi_delta_param::bda (C++ member), 208
 esp_bt_gap_cb_param_t::read_rssi_delta_param::stat (C++ member), 208
 esp_bt_gap_cb_param_t::remove_bond_dev (C++ member), 206
 esp_bt_gap_cb_param_t::rmt_srvc_rec (C++ member), 205
 esp_bt_gap_cb_param_t::rmt_srvc_rec_param (C++ class), 208
 esp_bt_gap_cb_param_t::rmt_srvc_rec_param (C++ member), 208
 esp_bt_gap_cb_param_t::rmt_srvc_rec_param::bda (C++ member), 208
 esp_bt_gap_cb_param_t::rmt_srvcs (C++ member), 205
 esp_bt_gap_cb_param_t::rmt_srvcs_param (C++ class), 209
 esp_bt_gap_cb_param_t::rmt_srvcs_param::bda (C++ member), 209
 esp_bt_gap_cb_param_t::rmt_srvcs_param::num_uuids (C++ member), 209
 esp_bt_gap_cb_param_t::rmt_srvcs_param::stat (C++ member), 209
 esp_bt_gap_cb_param_t::rmt_srvcs_param::supported_list (C++ member), 209
 esp_bt_gap_cb_param_t::set_afh_channels (C++ member), 206
 esp_bt_gap_cb_param_t::set_afh_channels_param (C++ class), 209
 esp_bt_gap_cb_param_t::set_afh_channels_param (C++ member), 209
 esp_bt_gap_cb_t (C++ type), 212
 ESP_BT_GAP_CFM_REQ_EVT (C++ enumerator), 215
 esp_bt_gap_config_eir_data (C++ function), 202
 ESP_BT_GAP_CONFIG_EIR_DATA_EVT (C++ enumerator), 215
 ESP_BT_GAP_DEV_PROP_BDNAME (C++ enumerator), 213
 ESP_BT_GAP_DEV_PROP_COD (C++ enumerator), 213
 ESP_BT_GAP_DEV_PROP_EIR (C++ enumerator), 213
 ESP_BT_GAP_DEV_PROP_RSSI (C++ enumerator), 213
 esp_bt_gap_dev_prop_t (C++ class), 209
 esp_bt_gap_dev_prop_t::len (C++ member), 209
 esp_bt_gap_dev_prop_t::type (C++ member), 209
 esp_bt_gap_dev_prop_t::val (C++ member), 209
 esp_bt_gap_dev_prop_type_t (C++ enum), 213
 ESP_BT_GAP_DEV_STATE_CHANGED_EVT (C++ enumerator), 214
 ESP_BT_GAP_DEV_STATE_CHANGED_EVT (C++ enumerator), 214
 ESP_BT_GAP_DISCOVERY_STARTED (C++ enumerator), 214
 esp_bt_gap_discovery_state_t (C++ enum), 214
 ESP_BT_GAP_DISCOVERY_STOPPED (C++ enumerator), 214
 ESP_BT_GAP_EIR_DATA_LEN (C macro), 210
 ESP_BT_GAP_EVT_MAX (C++ enumerator), 215
 esp_bt_gap_get_bond_device_list (C++ function), 203
 esp_bt_gap_get_bond_device_num (C++ function), 203
 esp_bt_gap_get_cod (C++ function), 203
 esp_bt_gap_get_cod_format_type (C++ function), 201
 esp_bt_gap_get_cod_major_dev (C++ function), 201
 esp_bt_gap_get_cod_minor_dev (C++ function), 201
 esp_bt_gap_get_cod_srvc (C++ function), 201
 esp_bt_gap_get_remote_service_record (C++ function), 202
 esp_bt_gap_get_remote_services (C++ function), 202
 esp_bt_gap_is_valid_cod (C++ function), 201
 ESP_BT_GAP_KEY_NOTIF_EVT (C++ enumerator), 215
 ESP_BT_GAP_KEY_REQ_EVT (C++ enumerator), 215
 stat
 ESP_BT_GAP_MAX_BDNAME_LEN (C macro), 210
 ESP_BT_GAP_MAX_INQ_LEN (C macro), 212
 ESP_BT_GAP_MIN_INQ_LEN (C macro), 212

- esp_bt_gap_pin_reply (C++ function), 204
 ESP_BT_GAP_PIN_REQ_EVT (C++ enumerator), 215
 ESP_BT_GAP_QOS_CMPL_EVT (C++ enumerator), 215
 esp_bt_gap_read_remote_name (C++ function), 205
 ESP_BT_GAP_READ_REMOTE_NAME_EVT (C++ enumerator), 215
 esp_bt_gap_read_rssi_delta (C++ function), 203
 ESP_BT_GAP_READ_RSSI_DELTA_EVT (C++ enumerator), 215
 esp_bt_gap_register_callback (C++ function), 201
 ESP_BT_GAP_REMOVE_BOND_DEV_COMPLETE_EVT (C++ enumerator), 215
 esp_bt_gap_remove_bond_device (C++ function), 203
 esp_bt_gap_resolve_eir_data (C++ function), 202
 ESP_BT_GAP_RMT_SRVC_REC_EVT (C++ enumerator), 214
 ESP_BT_GAP_RMT_SRVCS_EVT (C++ enumerator), 214
 ESP_BT_GAP_RSSI_HIGH_THRLD (C macro), 210
 ESP_BT_GAP_RSSI_LOW_THRLD (C macro), 210
 esp_bt_gap_set_afh_channels (C++ function), 204
 ESP_BT_GAP_SET_AFH_CHANNELS_EVT (C++ enumerator), 215
 esp_bt_gap_set_cod (C++ function), 203
 esp_bt_gap_set_pin (C++ function), 204
 esp_bt_gap_set_qos (C++ function), 205
 esp_bt_gap_set_scan_mode (C++ function), 201
 esp_bt_gap_set_security_param (C++ function), 204
 esp_bt_gap_ssp_confirm_reply (C++ function), 204
 esp_bt_gap_ssp_passkey_reply (C++ function), 204
 esp_bt_gap_start_discovery (C++ function), 201
 ESP_BT_GENERAL_DISCOVERABLE (C++ enumerator), 213
 esp_bt_hf_answer_call (C++ function), 268
 esp_bt_hf_bsir (C++ function), 268
 esp_bt_hf_cind_response (C++ function), 267
 esp_bt_hf_clcc_response (C++ function), 268
 ESP_BT_HF_CLIENT_NUMBER_LEN (C macro), 262
 ESP_BT_HF_CLIENT_OPERATOR_NAME_LEN (C macro), 262
 esp_bt_hf_cmee_response (C++ function), 266
 esp_bt_hf_cnum_response (C++ function), 268
 esp_bt_hf_connect (C++ function), 265
 esp_bt_hf_connect_audio (C++ function), 266
 esp_bt_hf_cops_response (C++ function), 267
 esp_bt_hf_deinit (C++ function), 265
 esp_bt_hf_disconnect (C++ function), 265
 esp_bt_hf_disconnect_audio (C++ function), 266
 esp_bt_hf_end_call (C++ function), 269
 esp_bt_hf_indchange_notification (C++ function), 267
 esp_bt_hf_init (C++ function), 265
 ESP_BT_HF_NUMBER_LEN (C macro), 246
 ESP_BT_HF_OPERATOR_NAME_LEN (C macro), 246
 esp_bt_hf_out_call (C++ function), 269
 esp_bt_hf_register_callback (C++ function), 265
 esp_bt_hf_register_data_callback (C++ function), 270
 esp_bt_hf_reject_call (C++ function), 269
 esp_bt_hf_volume_control (C++ function), 266
 esp_bt_hf_vra (C++ function), 266
 ESP_BT_INIT_COD (C++ enumerator), 212
 ESP_BT_INQ_MODE_GENERAL_INQUIRY (C++ enumerator), 215
 ESP_BT_INQ_MODE_LIMITED_INQUIRY (C++ enumerator), 215
 esp_bt_inq_mode_t (C++ enum), 215
 ESP_BT_IO_CAP_IN (C macro), 211
 ESP_BT_IO_CAP_IO (C macro), 211
 ESP_BT_IO_CAP_NONE (C macro), 211
 ESP_BT_IO_CAP_OUT (C macro), 211
 esp_bt_io_cap_t (C++ type), 212
 ESP_BT_LIMITED_DISCOVERABLE (C++ enumerator), 212
 esp_bt_mem_release (C++ function), 115
 ESP_BT_MODE_BLE (C++ enumerator), 118
 ESP_BT_MODE_BTDM (C++ enumerator), 118
 ESP_BT_MODE_CLASSIC_BT (C++ enumerator), 118
 ESP_BT_MODE_IDLE (C++ enumerator), 118
 esp_bt_mode_t (C++ enum), 118
 ESP_BT_NON_CONNECTABLE (C++ enumerator), 212
 ESP_BT_NON_DISCOVERABLE (C++ enumerator), 212
 ESP_BT_OCTET16_LEN (C macro), 121
 esp_bt_octet16_t (C++ type), 122
 ESP_BT_OCTET8_LEN (C macro), 121
 esp_bt_octet8_t (C++ type), 122
 ESP_BT_PIN_CODE_LEN (C macro), 211
 esp_bt_pin_code_t (C++ type), 212
 ESP_BT_PIN_TYPE_FIXED (C++ enumerator), 213
 esp_bt_pin_type_t (C++ enum), 213
 ESP_BT_PIN_TYPE_VARIABLE (C++ enumerator), 213
 ESP_BT_SET_COD_ALL (C++ enumerator), 212
 ESP_BT_SET_COD_MAJOR_MINOR (C++ enumerator), 212

- ESP_BT_SET_COD_SERVICE_CLASS (C++ *enumerator*), 122
- esp_bt_sleep_enable (C++ *function*), 116
- ESP_BT_SP_IOCAP_MODE (C++ *enumerator*), 213
- esp_bt_sp_param_t (C++ *enum*), 213
- ESP_BT_STATUS_AUTH_FAILURE (C++ *enumerator*), 122
- ESP_BT_STATUS_AUTH_REJECTED (C++ *enumerator*), 122
- ESP_BT_STATUS_BUSY (C++ *enumerator*), 122
- ESP_BT_STATUS_CONTROL_LE_DATA_LEN_UNSUPPORTED (C++ *function*), 1131 (C++ *enumerator*), 122
- ESP_BT_STATUS_DONE (C++ *enumerator*), 122
- ESP_BT_STATUS_EIR_TOO_LARGE (C++ *enumerator*), 122
- ESP_BT_STATUS_ERR_ILLEGAL_PARAMETER_FORMAT (C++ *enumerator*), 122
- ESP_BT_STATUS_FAIL (C++ *enumerator*), 122
- ESP_BT_STATUS_INVALID_STATIC_RAND_ADDR (C++ *enumerator*), 122
- ESP_BT_STATUS_MEMORY_FULL (C++ *enumerator*), 122
- ESP_BT_STATUS_NOMEM (C++ *enumerator*), 122
- ESP_BT_STATUS_NOT_READY (C++ *enumerator*), 122
- ESP_BT_STATUS_PARAM_OUT_OF_RANGE (C++ *enumerator*), 122
- ESP_BT_STATUS_PARM_INVALID (C++ *enumerator*), 122
- ESP_BT_STATUS_PEER_LE_DATA_LEN_UNSUPPORTED (C++ *enumerator*), 122
- ESP_BT_STATUS_PENDING (C++ *enumerator*), 122
- ESP_BT_STATUS_RMT_DEV_DOWN (C++ *enumerator*), 122
- ESP_BT_STATUS_SUCCESS (C++ *enumerator*), 122
- esp_bt_status_t (C++ *enum*), 122
- ESP_BT_STATUS_TIMEOUT (C++ *enumerator*), 122
- ESP_BT_STATUS_UNACCEPT_CONN_INTERVAL (C++ *enumerator*), 122
- ESP_BT_STATUS_UNHANDLED (C++ *enumerator*), 122
- ESP_BT_STATUS_UNSUPPORTED (C++ *enumerator*), 122
- esp_bt_uuid_t (C++ *class*), 120
- esp_bt_uuid_t::len (C++ *member*), 121
- esp_bt_uuid_t::uuid (C++ *member*), 121
- ESP_CHIP_ID_ESP32 (C++ *enumerator*), 1001
- ESP_CHIP_ID_INVALID (C++ *enumerator*), 1001
- esp_chip_id_t (C++ *enum*), 1001
- esp_chip_info (C++ *function*), 1176
- esp_chip_info_t (C++ *class*), 1176
- esp_chip_info_t::cores (C++ *member*), 1176
- esp_chip_info_t::features (C++ *member*), 1176
- esp_chip_info_t::model (C++ *member*), 1176
- esp_chip_info_t::revision (C++ *member*), 1176
- esp_chip_model_t (C++ *enum*), 1177
- esp_deep_sleep (C++ *function*), 1201
- esp_deep_sleep_disable_rom_logging (C++ *function*), 1202
- esp_deep_sleep_start (C++ *function*), 1201
- esp_deep_sleep_wake_stub_fn_t (C++ *type*), 1202
- ESP_DEFAULT_GATT_IF (C *macro*), 121
- esp_default_wake_deep_sleep (C++ *function*), 1202
- esp_deregister_freertos_idle_hook (C++ *function*), 1131
- esp_deregister_freertos_idle_hook_for_cpu (C++ *function*), 1131
- esp_deregister_freertos_tick_hook (C++ *function*), 1131
- esp_deregister_freertos_tick_hook_for_cpu (C++ *function*), 1131
- esp_derive_local_mac (C++ *function*), 1176
- esp_duplicate_info_t (C++ *type*), 146
- esp_duplicate_scan_exceptional_list_type_t (C++ *enum*), 151
- ESP_EARLY_LOGD (C *macro*), 1170
- ESP_EARLY_LOGE (C *macro*), 1170
- ESP_EARLY_LOGI (C *macro*), 1170
- ESP_EARLY_LOGV (C *macro*), 1170
- ESP_EARLY_LOGW (C *macro*), 1170
- esp_efuse_batch_write_begin (C++ *function*), 1017
- esp_efuse_batch_write_cancel (C++ *function*), 1017
- esp_efuse_batch_write_commit (C++ *function*), 1017
- esp_efuse_burn_new_values (C++ *function*), 1016
- esp_efuse_check_secure_version (C++ *function*), 1016
- esp_efuse_desc_t (C++ *class*), 1017
- esp_efuse_desc_t::bit_count (C++ *member*), 1017
- esp_efuse_desc_t::bit_start (C++ *member*), 1017
- esp_efuse_desc_t::efuse_block (C++ *member*), 1017
- esp_efuse_disable_basic_rom_console (C++ *function*), 1016
- esp_efuse_disable_rom_download_mode (C++ *function*), 1016
- esp_efuse_get_chip_ver (C++ *function*), 1016
- esp_efuse_get_coding_scheme (C++ *function*), 1016
- esp_efuse_get_field_size (C++ *function*), 1015
- esp_efuse_get_pkg_ver (C++ *function*), 1016
- esp_efuse_init (C++ *function*), 1017
- esp_efuse_mac_get_custom (C++ *function*), 1175
- esp_efuse_mac_get_default (C++ *function*), 1175

- esp_efuse_read_block (C++ function), 1016
 esp_efuse_read_field_bit (C++ function), 1014
 esp_efuse_read_field_blob (C++ function), 1013
 esp_efuse_read_field_cnt (C++ function), 1014
 esp_efuse_read_reg (C++ function), 1015
 esp_efuse_read_secure_version (C++ function), 1016
 esp_efuse_reset (C++ function), 1016
 esp_efuse_set_read_protect (C++ function), 1015
 esp_efuse_set_write_protect (C++ function), 1015
 esp_efuse_update_secure_version (C++ function), 1017
 esp_efuse_write_block (C++ function), 1016
 esp_efuse_write_field_bit (C++ function), 1014
 esp_efuse_write_field_blob (C++ function), 1014
 esp_efuse_write_field_cnt (C++ function), 1014
 esp_efuse_write_random_key (C++ function), 1016
 esp_efuse_write_reg (C++ function), 1015
 ESP_ERR_CODING (C macro), 1017
 ESP_ERR_EFUSE (C macro), 1017
 ESP_ERR_EFUSE_CNT_IS_FULL (C macro), 1017
 ESP_ERR_EFUSE_REPEATED_PROG (C macro), 1017
 ESP_ERR_ESP_TLS_BASE (C macro), 818
 ESP_ERR_ESP_TLS_CANNOT_CREATE_SOCKET (C macro), 818
 ESP_ERR_ESP_TLS_CANNOT_RESOLVE_HOSTNAME (C macro), 818
 ESP_ERR_ESP_TLS_CONNECTION_TIMEOUT (C macro), 819
 ESP_ERR_ESP_TLS_FAILED_CONNECT_TO_HOST (C macro), 818
 ESP_ERR_ESP_TLS_SOCKET_SETOPT_FAILED (C macro), 818
 ESP_ERR_ESP_TLS_UNSUPPORTED_PROTOCOL_FAMILY (C macro), 818
 ESP_ERR_ESPNOW_ARG (C macro), 529
 ESP_ERR_ESPNOW_BASE (C macro), 529
 ESP_ERR_ESPNOW_EXIST (C macro), 529
 ESP_ERR_ESPNOW_FULL (C macro), 529
 ESP_ERR_ESPNOW_IF (C macro), 529
 ESP_ERR_ESPNOW_INTERNAL (C macro), 529
 ESP_ERR_ESPNOW_NO_MEM (C macro), 529
 ESP_ERR_ESPNOW_NOT_FOUND (C macro), 529
 ESP_ERR_ESPNOW_NOT_INIT (C macro), 529
 ESP_ERR_FLASH_BASE (C macro), 1019
 ESP_ERR_HTTP_BASE (C macro), 829
 ESP_ERR_HTTP_CONNECT (C macro), 829
 ESP_ERR_HTTP_CONNECTING (C macro), 830
 ESP_ERR_HTTP_EAGAIN (C macro), 830
 ESP_ERR_HTTP_FETCH_HEADER (C macro), 830
 ESP_ERR_HTTP_INVALID_TRANSPORT (C macro), 830
 ESP_ERR_HTTP_MAX_REDIRECT (C macro), 829
 ESP_ERR_HTTP_WRITE_DATA (C macro), 829
 ESP_ERR_HTTPD_ALLOC_MEM (C macro), 849
 ESP_ERR_HTTPD_BASE (C macro), 849
 ESP_ERR_HTTPD_HANDLER_EXISTS (C macro), 849
 ESP_ERR_HTTPD_HANDLERS_FULL (C macro), 849
 ESP_ERR_HTTPD_INVALID_REQ (C macro), 849
 ESP_ERR_HTTPD_RESP_HDR (C macro), 849
 ESP_ERR_HTTPD_RESP_SEND (C macro), 849
 ESP_ERR_HTTPD_RESULT_TRUNC (C macro), 849
 ESP_ERR_HTTPD_TASK (C macro), 849
 ESP_ERR_HTTPS_OTA_BASE (C macro), 1021
 ESP_ERR_HTTPS_OTA_IN_PROGRESS (C macro), 1021
 ESP_ERR_INVALID_ARG (C macro), 1018
 ESP_ERR_INVALID_CRC (C macro), 1018
 ESP_ERR_INVALID_MAC (C macro), 1018
 ESP_ERR_INVALID_RESPONSE (C macro), 1018
 ESP_ERR_INVALID_SIZE (C macro), 1018
 ESP_ERR_INVALID_STATE (C macro), 1018
 ESP_ERR_INVALID_VERSION (C macro), 1018
 ESP_ERR_MBEDTLS_CERT_PARTLY_OK (C macro), 818
 ESP_ERR_MBEDTLS_CTR_DRBG_SEED_FAILED (C macro), 818
 ESP_ERR_MBEDTLS_PK_PARSE_KEY_FAILED (C macro), 819
 ESP_ERR_MBEDTLS_SSL_CONF_ALPN_PROTOCOLS_FAILED (C macro), 818
 ESP_ERR_MBEDTLS_SSL_CONF_OWN_CERT_FAILED (C macro), 819
 ESP_ERR_MBEDTLS_SSL_CONF_PSK_FAILED (C macro), 819
 ESP_ERR_MBEDTLS_SSL_CONFIG_DEFAULTS_FAILED (C macro), 818
 ESP_ERR_MBEDTLS_SSL_HANDSHAKE_FAILED (C macro), 819
 ESP_ERR_MBEDTLS_SSL_SET_HOSTNAME_FAILED (C macro), 818
 ESP_ERR_MBEDTLS_SSL_SETUP_FAILED (C macro), 819
 ESP_ERR_MBEDTLS_SSL_WRITE_FAILED (C macro), 819
 ESP_ERR_MBEDTLS_X509_CRT_PARSE_FAILED (C macro), 819
 ESP_ERR_MESH_ARGUMENT (C macro), 554
 ESP_ERR_MESH_BASE (C macro), 1019
 ESP_ERR_MESH_DISCARD (C macro), 555
 ESP_ERR_MESH_DISCARD_DUPLICATE (C macro), 555
 ESP_ERR_MESH_DISCONNECTED (C macro), 554
 ESP_ERR_MESH_EXCEED_MTU (C macro), 554

- ESP_ERR_MESH_INTERFACE (*C macro*), 555
- ESP_ERR_MESH_NO_MEMORY (*C macro*), 554
- ESP_ERR_MESH_NO_PARENT_FOUND (*C macro*), 554
- ESP_ERR_MESH_NO_ROUTE_FOUND (*C macro*), 554
- ESP_ERR_MESH_NOT_ALLOWED (*C macro*), 554
- ESP_ERR_MESH_NOT_CONFIG (*C macro*), 554
- ESP_ERR_MESH_NOT_INIT (*C macro*), 554
- ESP_ERR_MESH_NOT_START (*C macro*), 554
- ESP_ERR_MESH_NOT_SUPPORT (*C macro*), 554
- ESP_ERR_MESH_OPTION_NULL (*C macro*), 554
- ESP_ERR_MESH_OPTION_UNKNOWN (*C macro*), 554
- ESP_ERR_MESH_QUEUE_FAIL (*C macro*), 554
- ESP_ERR_MESH_QUEUE_FULL (*C macro*), 554
- ESP_ERR_MESH_QUEUE_READ (*C macro*), 555
- ESP_ERR_MESH_RECV_RELEASE (*C macro*), 555
- ESP_ERR_MESH_TIMEOUT (*C macro*), 554
- ESP_ERR_MESH_VOTING (*C macro*), 555
- ESP_ERR_MESH_WIFI_NOT_START (*C macro*), 554
- ESP_ERR_MESH_XMIT (*C macro*), 555
- ESP_ERR_MESH_XON_NO_WINDOW (*C macro*), 555
- ESP_ERR_NO_MEM (*C macro*), 1018
- ESP_ERR_NOT_FINISHED (*C macro*), 889
- ESP_ERR_NOT_FOUND (*C macro*), 1018
- ESP_ERR_NOT_SUPPORTED (*C macro*), 1018
- ESP_ERR_NVS_BASE (*C macro*), 944
- ESP_ERR_NVS_CONTENT_DIFFERS (*C macro*), 945
- ESP_ERR_NVS_CORRUPT_KEY_PART (*C macro*), 945
- ESP_ERR_NVS_ENCR_NOT_SUPPORTED (*C macro*), 945
- ESP_ERR_NVS_INVALID_HANDLE (*C macro*), 944
- ESP_ERR_NVS_INVALID_LENGTH (*C macro*), 945
- ESP_ERR_NVS_INVALID_NAME (*C macro*), 944
- ESP_ERR_NVS_INVALID_STATE (*C macro*), 945
- ESP_ERR_NVS_KEY_TOO_LONG (*C macro*), 945
- ESP_ERR_NVS_KEYS_NOT_INITIALIZED (*C macro*), 945
- ESP_ERR_NVS_NEW_VERSION_FOUND (*C macro*), 945
- ESP_ERR_NVS_NO_FREE_PAGES (*C macro*), 945
- ESP_ERR_NVS_NOT_ENOUGH_SPACE (*C macro*), 944
- ESP_ERR_NVS_NOT_FOUND (*C macro*), 944
- ESP_ERR_NVS_NOT_INITIALIZED (*C macro*), 944
- ESP_ERR_NVS_PAGE_FULL (*C macro*), 945
- ESP_ERR_NVS_PART_NOT_FOUND (*C macro*), 945
- ESP_ERR_NVS_READ_ONLY (*C macro*), 944
- ESP_ERR_NVS_REMOVE_FAILED (*C macro*), 944
- ESP_ERR_NVS_TYPE_MISMATCH (*C macro*), 944
- ESP_ERR_NVS_VALUE_TOO_LONG (*C macro*), 945
- ESP_ERR_NVS_XTS_CFG_FAILED (*C macro*), 945
- ESP_ERR_NVS_XTS_CFG_NOT_FOUND (*C macro*), 945
- ESP_ERR_NVS_XTS_DECR_FAILED (*C macro*), 945
- ESP_ERR_NVS_XTS_ENCR_FAILED (*C macro*), 945
- ESP_ERR_OTA_BASE (*C macro*), 1187
- ESP_ERR_OTA_PARTITION_CONFLICT (*C macro*), 1187
- ESP_ERR_OTA_ROLLBACK_FAILED (*C macro*), 1187
- ESP_ERR_OTA_ROLLBACK_INVALID_STATE (*C macro*), 1187
- ESP_ERR_OTA_SELECT_INFO_INVALID (*C macro*), 1187
- ESP_ERR_OTA_SMALL_SEC_VER (*C macro*), 1187
- ESP_ERR_OTA_VALIDATE_FAILED (*C macro*), 1187
- esp_err_t (*C++ type*), 1019
- ESP_ERR_TIMEOUT (*C macro*), 1018
- esp_err_to_name (*C++ function*), 1018
- esp_err_to_name_r (*C++ function*), 1018
- ESP_ERR_ULP_BASE (*C macro*), 1669
- ESP_ERR_ULP_BRANCH_OUT_OF_RANGE (*C macro*), 1670
- ESP_ERR_ULP_DUPLICATE_LABEL (*C macro*), 1670
- ESP_ERR_ULP_INVALID_LOAD_ADDR (*C macro*), 1670
- ESP_ERR_ULP_SIZE_TOO_BIG (*C macro*), 1669
- ESP_ERR_ULP_UNDEFINED_LABEL (*C macro*), 1670
- ESP_ERR_WIFI_BASE (*C macro*), 1018
- ESP_ERR_WIFI_CONN (*C macro*), 502
- ESP_ERR_WIFI_IF (*C macro*), 502
- ESP_ERR_WIFI_INIT_STATE (*C macro*), 503
- ESP_ERR_WIFI_MAC (*C macro*), 502
- ESP_ERR_WIFI_MODE (*C macro*), 502
- ESP_ERR_WIFI_NOT_CONNECT (*C macro*), 503
- ESP_ERR_WIFI_NOT_INIT (*C macro*), 502
- ESP_ERR_WIFI_NOT_STARTED (*C macro*), 502
- ESP_ERR_WIFI_NOT_STOPPED (*C macro*), 502
- ESP_ERR_WIFI_NVS (*C macro*), 502
- ESP_ERR_WIFI_PASSWORD (*C macro*), 503
- ESP_ERR_WIFI_POST (*C macro*), 503
- ESP_ERR_WIFI_SSID (*C macro*), 503
- ESP_ERR_WIFI_STATE (*C macro*), 502
- ESP_ERR_WIFI_STOP_STATE (*C macro*), 503
- ESP_ERR_WIFI_TIMEOUT (*C macro*), 503
- ESP_ERR_WIFI_WAKE_FAIL (*C macro*), 503
- ESP_ERR_WIFI_WOULD_BLOCK (*C macro*), 503
- ESP_ERROR_CHECK (*C macro*), 1019
- ESP_ERROR_CHECK_WITHOUT_ABORT (*C macro*), 1019
- esp_esptouch_set_timeout (*C++ function*), 522
- esp_eth_clear_default_handlers (*C++ function*), 573

- esp_eth_config_t (C++ class), 562
 esp_eth_config_t::check_link_period_ms (C++ member), 562
 esp_eth_config_t::mac (C++ member), 562
 esp_eth_config_t::on_lowlevel_deinit_done (C++ member), 562
 esp_eth_config_t::on_lowlevel_init_done (C++ member), 562
 esp_eth_config_t::phy (C++ member), 562
 esp_eth_config_t::stack_input (C++ member), 562
 esp_eth_decrease_reference (C++ function), 561
 esp_eth_del_netif_glue (C++ function), 572
 esp_eth_detect_phy_addr (C++ function), 563
 esp_eth_driver_install (C++ function), 560
 esp_eth_driver_uninstall (C++ function), 560
 esp_eth_handle_t (C++ type), 562
 esp_eth_increase_reference (C++ function), 561
 esp_eth_io_cmd_t (C++ enum), 564
 esp_eth_ioctl (C++ function), 561
 esp_eth_mac_s (C++ class), 566
 esp_eth_mac_s::deinit (C++ member), 566
 esp_eth_mac_s::del (C++ member), 568
 esp_eth_mac_s::get_addr (C++ member), 568
 esp_eth_mac_s::init (C++ member), 566
 esp_eth_mac_s::read_phy_reg (C++ member), 567
 esp_eth_mac_s::receive (C++ member), 567
 esp_eth_mac_s::set_addr (C++ member), 567
 esp_eth_mac_s::set_duplex (C++ member), 568
 esp_eth_mac_s::set_link (C++ member), 568
 esp_eth_mac_s::set_mediator (C++ member), 566
 esp_eth_mac_s::set_promiscuous (C++ member), 568
 esp_eth_mac_s::set_speed (C++ member), 568
 esp_eth_mac_s::start (C++ member), 566
 esp_eth_mac_s::stop (C++ member), 566
 esp_eth_mac_s::transmit (C++ member), 566
 esp_eth_mac_s::write_phy_reg (C++ member), 567
 esp_eth_mac_t (C++ type), 569
 esp_eth_mediator_s (C++ class), 563
 esp_eth_mediator_s::on_state_changed (C++ member), 564
 esp_eth_mediator_s::phy_reg_read (C++ member), 563
 esp_eth_mediator_s::phy_reg_write (C++ member), 563
 esp_eth_mediator_s::stack_input (C++ member), 563
 esp_eth_mediator_t (C++ type), 564
 esp_eth_new_netif_glue (C++ function), 572
 ESP_ETH_PHY_ADDR_AUTO (C macro), 572
 esp_eth_phy_new_dp83848 (C++ function), 570
 esp_eth_phy_new_ip101 (C++ function), 569
 esp_eth_phy_new_lan8720 (C++ function), 570
 esp_eth_phy_new_rtl8201 (C++ function), 569
 esp_eth_phy_s (C++ class), 570
 esp_eth_phy_s::deinit (C++ member), 571
 esp_eth_phy_s::del (C++ member), 571
 esp_eth_phy_s::get_addr (C++ member), 571
 esp_eth_phy_s::get_link (C++ member), 571
 esp_eth_phy_s::init (C++ member), 570
 esp_eth_phy_s::negotiate (C++ member), 571
 esp_eth_phy_s::pwrctl (C++ member), 571
 esp_eth_phy_s::reset (C++ member), 570
 esp_eth_phy_s::reset_hw (C++ member), 570
 esp_eth_phy_s::set_addr (C++ member), 571
 esp_eth_phy_s::set_mediator (C++ member), 570
 esp_eth_phy_t (C++ type), 572
 esp_eth_receive (C++ function), 561
 esp_eth_set_default_handlers (C++ function), 572
 esp_eth_start (C++ function), 560
 esp_eth_state_t (C++ enum), 564
 esp_eth_stop (C++ function), 560
 esp_eth_transmit (C++ function), 561
 esp_eth_update_input_path (C++ function), 560
 ESP_EVENT_ANY_BASE (C macro), 1032
 ESP_EVENT_ANY_ID (C macro), 1032
 esp_event_base_t (C++ type), 1032
 ESP_EVENT_DECLARE_BASE (C macro), 1032
 ESP_EVENT_DEFINE_BASE (C macro), 1032
 esp_event_dump (C++ function), 1031
 esp_event_handler_register (C++ function), 1028
 esp_event_handler_register_with (C++ function), 1029
 esp_event_handler_t (C++ type), 1032
 esp_event_handler_unregister (C++ function), 1029
 esp_event_handler_unregister_with (C++ function), 1030
 esp_event_loop_args_t (C++ class), 1031
 esp_event_loop_args_t::queue_size (C++ member), 1031
 esp_event_loop_args_t::task_core_id (C++ member), 1032
 esp_event_loop_args_t::task_name (C++ member), 1031
 esp_event_loop_args_t::task_priority (C++ member), 1031
 esp_event_loop_args_t::task_stack_size (C++ member), 1031
 esp_event_loop_create (C++ function), 1027
 esp_event_loop_create_default (C++ function), 1028

- esp_event_loop_delete (C++ function), 1028
 esp_event_loop_delete_default (C++ function), 1028
 esp_event_loop_handle_t (C++ type), 1032
 esp_event_loop_init (C++ function), 1033
 esp_event_loop_run (C++ function), 1028
 esp_event_loop_set_cb (C++ function), 1033
 esp_event_post (C++ function), 1030
 esp_event_post_to (C++ function), 1030
 esp_event_process_default (C++ function), 1033
 esp_event_send (C++ function), 1032
 esp_event_send_internal (C++ function), 1032
 esp_event_set_default_eth_handlers (C++ function), 1033
 esp_event_set_default_wifi_handlers (C++ function), 1033
 ESP_EXECUTE_EXPRESSION_WITH_STACK (C macro), 1161
 esp_execute_shared_stack_function (C++ function), 1160
 ESP_EXT1_WAKEUP_ALL_LOW (C++ enumerator), 1202
 ESP_EXT1_WAKEUP_ANY_HIGH (C++ enumerator), 1202
 ESP_FAIL (C macro), 1018
 esp_fill_random (C++ function), 1175
 ESP_FLASH_10MHZ (C++ enumerator), 974
 ESP_FLASH_20MHZ (C++ enumerator), 974
 ESP_FLASH_26MHZ (C++ enumerator), 974
 ESP_FLASH_40MHZ (C++ enumerator), 974
 ESP_FLASH_5MHZ (C++ enumerator), 973
 ESP_FLASH_80MHZ (C++ enumerator), 974
 esp_flash_chip_driver_initialized (C++ function), 968
 ESP_FLASH_ENC_MODE_DEVELOPMENT (C++ enumerator), 981
 ESP_FLASH_ENC_MODE_DISABLED (C++ enumerator), 981
 ESP_FLASH_ENC_MODE_RELEASE (C++ enumerator), 981
 esp_flash_enc_mode_t (C++ enum), 981
 esp_flash_encrypt_check_and_update (C++ function), 980
 esp_flash_encrypt_region (C++ function), 980
 esp_flash_encryption_enabled (C++ function), 980
 esp_flash_encryption_init_checks (C++ function), 981
 esp_flash_erase_chip (C++ function), 968
 esp_flash_erase_region (C++ function), 968
 esp_flash_get_chip_write_protect (C++ function), 968
 esp_flash_get_protectable_regions (C++ function), 969
 esp_flash_get_protected_region (C++ function), 969
 esp_flash_get_size (C++ function), 968
 esp_flash_init (C++ function), 967
 esp_flash_io_mode_t (C++ enum), 974
 esp_flash_is_quad_mode (C++ function), 970
 esp_flash_os_functions_t (C++ class), 971
 esp_flash_os_functions_t::delay_us (C++ member), 971
 esp_flash_os_functions_t::end (C++ member), 971
 esp_flash_os_functions_t::region_protected (C++ member), 971
 esp_flash_os_functions_t::start (C++ member), 971
 esp_flash_os_functions_t::yield (C++ member), 971
 esp_flash_read (C++ function), 969
 esp_flash_read_encrypted (C++ function), 970
 esp_flash_read_id (C++ function), 968
 esp_flash_region_t (C++ class), 971
 esp_flash_region_t::offset (C++ member), 971
 esp_flash_region_t::size (C++ member), 971
 esp_flash_set_chip_write_protect (C++ function), 968
 esp_flash_set_protected_region (C++ function), 969
 ESP_FLASH_SPEED_MAX (C++ enumerator), 974
 ESP_FLASH_SPEED_MIN (C macro), 973
 esp_flash_speed_t (C++ enum), 973
 esp_flash_spi_device_config_t (C++ class), 967
 esp_flash_spi_device_config_t::cs_id (C++ member), 967
 esp_flash_spi_device_config_t::cs_io_num (C++ member), 967
 esp_flash_spi_device_config_t::host_id (C++ member), 967
 esp_flash_spi_device_config_t::input_delay_ns (C++ member), 967
 esp_flash_spi_device_config_t::io_mode (C++ member), 967
 esp_flash_spi_device_config_t::speed (C++ member), 967
 esp_flash_t (C++ class), 971
 esp_flash_t (C++ type), 972
 esp_flash_t::chip_drv (C++ member), 971
 esp_flash_t::chip_id (C++ member), 971
 esp_flash_t::host (C++ member), 971
 esp_flash_t::os_func (C++ member), 971
 esp_flash_t::os_func_data (C++ member), 971
 esp_flash_t::read_mode (C++ member), 971
 esp_flash_t::size (C++ member), 971
 esp_flash_write (C++ function), 970
 esp_flash_write_encrypted (C++ function),

- 970
- esp_flash_write_protect_crypt_cnt (C++ function), 980
- esp_freertos_idle_cb_t (C++ type), 1131
- esp_freertos_tick_cb_t (C++ type), 1131
- ESP_GAP_BLE_ADD_WHITELIST_COMPLETE_EVT (C macro), 145
- ESP_GAP_BLE_ADV_DATA_RAW_SET_COMPLETE_EVT (C++ enumerator), 146
- ESP_GAP_BLE_ADV_DATA_SET_COMPLETE_EVT (C++ enumerator), 146
- ESP_GAP_BLE_ADV_START_COMPLETE_EVT (C++ enumerator), 146
- ESP_GAP_BLE_ADV_STOP_COMPLETE_EVT (C++ enumerator), 147
- ESP_GAP_BLE_AUTH_CMPL_EVT (C++ enumerator), 146
- esp_gap_ble_cb_event_t (C++ enum), 146
- esp_gap_ble_cb_t (C++ type), 146
- esp_gap_ble_channels (C++ type), 146
- ESP_GAP_BLE_CHANNELS_LEN (C macro), 145
- ESP_GAP_BLE_CLEAR_BOND_DEV_COMPLETE_EVT (C++ enumerator), 147
- ESP_GAP_BLE_EVT_MAX (C++ enumerator), 147
- ESP_GAP_BLE_GET_BOND_DEV_COMPLETE_EVT (C++ enumerator), 147
- ESP_GAP_BLE_KEY_EVT (C++ enumerator), 146
- ESP_GAP_BLE_LOCAL_ER_EVT (C++ enumerator), 147
- ESP_GAP_BLE_LOCAL_IR_EVT (C++ enumerator), 146
- ESP_GAP_BLE_NC_REQ_EVT (C++ enumerator), 147
- ESP_GAP_BLE_OOB_REQ_EVT (C++ enumerator), 146
- ESP_GAP_BLE_PASSKEY_NOTIF_EVT (C++ enumerator), 146
- ESP_GAP_BLE_PASSKEY_REQ_EVT (C++ enumerator), 146
- ESP_GAP_BLE_READ_RSSI_COMPLETE_EVT (C++ enumerator), 147
- ESP_GAP_BLE_REMOVE_BOND_DEV_COMPLETE_EVT (C++ enumerator), 147
- ESP_GAP_BLE_SCAN_PARAM_SET_COMPLETE_EVT (C++ enumerator), 146
- ESP_GAP_BLE_SCAN_RESULT_EVT (C++ enumerator), 146
- ESP_GAP_BLE_SCAN_RSP_DATA_RAW_SET_COMPLETE_EVT (C++ enumerator), 146
- ESP_GAP_BLE_SCAN_RSP_DATA_SET_COMPLETE_EVT (C++ enumerator), 146
- ESP_GAP_BLE_SCAN_START_COMPLETE_EVT (C++ enumerator), 146
- ESP_GAP_BLE_SCAN_STOP_COMPLETE_EVT (C++ enumerator), 147
- ESP_GAP_BLE_SEC_REQ_EVT (C++ enumerator), 146
- esp_gap_ble_set_authorization (C++ function), 131
- esp_gap_ble_set_channels (C++ function), 131
- ESP_GAP_BLE_SET_CHANNELS_EVT (C++ enumerator), 147
- ESP_GAP_BLE_SET_LOCAL_PRIVACY_COMPLETE_EVT (C++ enumerator), 147
- ESP_GAP_BLE_SET_PKT_LENGTH_COMPLETE_EVT (C++ enumerator), 147
- ESP_GAP_BLE_SET_STATIC_RAND_ADDR_EVT (C++ enumerator), 147
- ESP_GAP_BLE_UPDATE_CONN_PARAMS_EVT (C++ enumerator), 147
- ESP_GAP_BLE_UPDATE_DUPLICATE_EXCEPTIONAL_LIST_COMPLETED_EVT (C++ enumerator), 147
- ESP_GAP_BLE_UPDATE_WHITELIST_COMPLETE_EVT (C++ enumerator), 147
- esp_gap_conn_params_t (C++ class), 139
- esp_gap_conn_params_t::interval (C++ member), 139
- esp_gap_conn_params_t::latency (C++ member), 139
- esp_gap_conn_params_t::timeout (C++ member), 139
- ESP_GAP_SEARCH_DI_DISC_CMPL_EVT (C++ enumerator), 150
- ESP_GAP_SEARCH_DISC_BLE_RES_EVT (C++ enumerator), 150
- ESP_GAP_SEARCH_DISC_CMPL_EVT (C++ enumerator), 150
- ESP_GAP_SEARCH_DISC_RES_EVT (C++ enumerator), 150
- esp_gap_search_evt_t (C++ enum), 150
- ESP_GAP_SEARCH_INQ_CMPL_EVT (C++ enumerator), 150
- ESP_GAP_SEARCH_INQ_DISCARD_NUM_EVT (C++ enumerator), 150
- ESP_GAP_SEARCH_INQ_RES_EVT (C++ enumerator), 150
- ESP_GAP_SEARCH_SEARCH_CANCEL_CMPL_EVT (C++ enumerator), 150
- ESP_GATT_ALREADY_OPEN (C++ enumerator), 161
- ESP_GATT_APP_RSP (C++ enumerator), 161
- ESP_GATT_ATTR_HANDLE_MAX (C macro), 159
- ESP_GATT_AUTH_FAIL (C++ enumerator), 161
- ESP_GATT_AUTH_REQ_MITM (C++ enumerator), 162
- ESP_GATT_AUTH_REQ_NO_MITM (C++ enumerator), 162
- ESP_GATT_AUTH_REQ_NONE (C++ enumerator), 161
- ESP_GATT_AUTH_REQ_SIGNED_MITM (C++ enumerator), 162
- ESP_GATT_AUTH_REQ_SIGNED_NO_MITM (C++ enumerator), 162
- esp_gatt_auth_req_t (C++ enum), 161
- ESP_GATT_AUTO_RSP (C macro), 159
- ESP_GATT_BODY_SENSOR_LOCATION (C macro),

- 158
- ESP_GATT_BUSY (C++ enumerator), 160
- ESP_GATT_CANCEL (C++ enumerator), 161
- ESP_GATT_CCC_CFG_ERR (C++ enumerator), 161
- ESP_GATT_CHAR_PROP_BIT_AUTH (C macro), 159
- ESP_GATT_CHAR_PROP_BIT_BROADCAST (C macro), 159
- ESP_GATT_CHAR_PROP_BIT_EXT_PROP (C macro), 159
- ESP_GATT_CHAR_PROP_BIT_INDICATE (C macro), 159
- ESP_GATT_CHAR_PROP_BIT_NOTIFY (C macro), 159
- ESP_GATT_CHAR_PROP_BIT_READ (C macro), 159
- ESP_GATT_CHAR_PROP_BIT_WRITE (C macro), 159
- ESP_GATT_CHAR_PROP_BIT_WRITE_NR (C macro), 159
- esp_gatt_char_prop_t (C++ type), 160
- ESP_GATT_CMD_STARTED (C++ enumerator), 160
- ESP_GATT_CONGESTED (C++ enumerator), 161
- ESP_GATT_CONN_CONN_CANCEL (C++ enumerator), 161
- ESP_GATT_CONN_FAIL_ESTABLISH (C++ enumerator), 161
- ESP_GATT_CONN_L2C_FAILURE (C++ enumerator), 161
- ESP_GATT_CONN_LMP_TIMEOUT (C++ enumerator), 161
- ESP_GATT_CONN_NONE (C++ enumerator), 161
- esp_gatt_conn_params_t (C++ class), 154
- esp_gatt_conn_params_t::interval (C++ member), 154
- esp_gatt_conn_params_t::latency (C++ member), 154
- esp_gatt_conn_params_t::timeout (C++ member), 154
- esp_gatt_conn_reason_t (C++ enum), 161
- ESP_GATT_CONN_TERMINATE_LOCAL_HOST (C++ enumerator), 161
- ESP_GATT_CONN_TERMINATE_PEER_USER (C++ enumerator), 161
- ESP_GATT_CONN_TIMEOUT (C++ enumerator), 161
- ESP_GATT_CONN_UNKNOWN (C++ enumerator), 161
- ESP_GATT_DB_ALL (C++ enumerator), 162
- esp_gatt_db_attr_type_t (C++ enum), 162
- ESP_GATT_DB_CHARACTERISTIC (C++ enumerator), 162
- ESP_GATT_DB_DESCRIPTOR (C++ enumerator), 162
- ESP_GATT_DB_FULL (C++ enumerator), 160
- ESP_GATT_DB_INCLUDED_SERVICE (C++ enumerator), 162
- ESP_GATT_DB_PRIMARY_SERVICE (C++ enumerator), 162
- ESP_GATT_DB_SECONDARY_SERVICE (C++ enumerator), 162
- ESP_GATT_DUP_REG (C++ enumerator), 161
- ESP_GATT_ENCRYPTED_MITM (C++ enumerator), 161
- ESP_GATT_ENCRYPTED_NO_MITM (C++ enumerator), 161
- ESP_GATT_ERR_UNLIKELY (C++ enumerator), 160
- ESP_GATT_ERROR (C++ enumerator), 160
- ESP_GATT_HEART_RATE_CNTL_POINT (C macro), 158
- ESP_GATT_HEART_RATE_MEAS (C macro), 158
- esp_gatt_id_t (C++ class), 152
- esp_gatt_id_t::inst_id (C++ member), 152
- esp_gatt_id_t::uuid (C++ member), 152
- ESP_GATT_IF_NONE (C macro), 159
- esp_gatt_if_t (C++ type), 160
- ESP_GATT_ILLEGAL_HANDLE (C macro), 159
- ESP_GATT_ILLEGAL_PARAMETER (C++ enumerator), 160
- ESP_GATT_ILLEGAL_UUID (C macro), 159
- ESP_GATT_INSUF_AUTHENTICATION (C++ enumerator), 160
- ESP_GATT_INSUF_AUTHORIZATION (C++ enumerator), 160
- ESP_GATT_INSUF_ENCRYPTION (C++ enumerator), 160
- ESP_GATT_INSUF_KEY_SIZE (C++ enumerator), 160
- ESP_GATT_INSUF_RESOURCE (C++ enumerator), 160
- ESP_GATT_INTERNAL_ERROR (C++ enumerator), 160
- ESP_GATT_INVALID_ATTR_LEN (C++ enumerator), 160
- ESP_GATT_INVALID_CFG (C++ enumerator), 161
- ESP_GATT_INVALID_HANDLE (C++ enumerator), 160
- ESP_GATT_INVALID_OFFSET (C++ enumerator), 160
- ESP_GATT_INVALID_PDU (C++ enumerator), 160
- ESP_GATT_MAX_ATTR_LEN (C macro), 159
- ESP_GATT_MAX_READ_MULTI_HANDLES (C macro), 159
- ESP_GATT_MORE (C++ enumerator), 161
- ESP_GATT_NO_RESOURCES (C++ enumerator), 160
- ESP_GATT_NOT_ENCRYPTED (C++ enumerator), 161
- ESP_GATT_NOT_FOUND (C++ enumerator), 160
- ESP_GATT_NOT_LONG (C++ enumerator), 160
- ESP_GATT_OK (C++ enumerator), 160
- ESP_GATT_OUT_OF_RANGE (C++ enumerator), 161
- ESP_GATT_PENDING (C++ enumerator), 161
- ESP_GATT_PERM_READ (C macro), 159
- ESP_GATT_PERM_READ_AUTHORIZATION (C macro), 159
- ESP_GATT_PERM_READ_ENC_MITM (C macro), 159
- ESP_GATT_PERM_READ_ENCRYPTED (C macro),

- 159
- `esp_gatt_perm_t` (C++ type), 160
- `ESP_GATT_PERM_WRITE` (C macro), 159
- `ESP_GATT_PERM_WRITE_AUTHORIZATION` (C macro), 159
- `ESP_GATT_PERM_WRITE_ENC_MITM` (C macro), 159
- `ESP_GATT_PERM_WRITE_ENCRYPTED` (C macro), 159
- `ESP_GATT_PERM_WRITE_SIGNED` (C macro), 159
- `ESP_GATT_PERM_WRITE_SIGNED_MITM` (C macro), 159
- `ESP_GATT_PRC_IN_PROGRESS` (C++ enumerator), 161
- `ESP_GATT_PREP_WRITE_CANCEL` (C macro), 173
- `ESP_GATT_PREP_WRITE_CANCEL` (C++ enumerator), 160
- `ESP_GATT_PREP_WRITE_EXEC` (C macro), 173
- `ESP_GATT_PREP_WRITE_EXEC` (C++ enumerator), 160
- `esp_gatt_prep_write_type` (C++ enum), 160
- `ESP_GATT_PREPARE_Q_FULL` (C++ enumerator), 160
- `ESP_GATT_READ_NOT_PERMIT` (C++ enumerator), 160
- `ESP_GATT_REQ_NOT_SUPPORTED` (C++ enumerator), 160
- `ESP_GATT_RSP_BY_APP` (C macro), 159
- `esp_gatt_rsp_t` (C++ union), 152
- `esp_gatt_rsp_t::attr_value` (C++ member), 152
- `esp_gatt_rsp_t::handle` (C++ member), 152
- `ESP_GATT_SERVICE_FROM_NVS_FLASH` (C++ enumerator), 162
- `ESP_GATT_SERVICE_FROM_REMOTE_DEVICE` (C++ enumerator), 162
- `ESP_GATT_SERVICE_FROM_UNKNOWN` (C++ enumerator), 162
- `ESP_GATT_SERVICE_STARTED` (C++ enumerator), 161
- `esp_gatt_srvc_id_t` (C++ class), 152
- `esp_gatt_srvc_id_t::id` (C++ member), 153
- `esp_gatt_srvc_id_t::is_primary` (C++ member), 153
- `ESP_GATT_STACK_RSP` (C++ enumerator), 161
- `esp_gatt_status_t` (C++ enum), 160
- `ESP_GATT_UNKNOWN_ERROR` (C++ enumerator), 161
- `ESP_GATT_UNSUPPORT_GRP_TYPE` (C++ enumerator), 160
- `ESP_GATT_UUID_ALERT_LEVEL` (C macro), 157
- `ESP_GATT_UUID_ALERT_NTF_SVC` (C macro), 156
- `ESP_GATT_UUID_ALERT_STATUS` (C macro), 158
- `ESP_GATT_UUID_Automation_IO_SVC` (C macro), 157
- `ESP_GATT_UUID_BATTERY_LEVEL` (C macro), 158
- `ESP_GATT_UUID_BATTERY_SERVICE_SVC` (C macro), 156
- `ESP_GATT_UUID_BLOOD_PRESSURE_SVC` (C macro), 156
- `ESP_GATT_UUID_BODY_COMPOSITION` (C macro), 157
- `ESP_GATT_UUID_BOND_MANAGEMENT_SVC` (C macro), 157
- `ESP_GATT_UUID_CHAR_AGG_FORMAT` (C macro), 157
- `ESP_GATT_UUID_CHAR_CLIENT_CONFIG` (C macro), 157
- `ESP_GATT_UUID_CHAR_DECLARE` (C macro), 157
- `ESP_GATT_UUID_CHAR_DESCRIPTION` (C macro), 157
- `ESP_GATT_UUID_CHAR_EXT_PROP` (C macro), 157
- `ESP_GATT_UUID_CHAR_PRESENT_FORMAT` (C macro), 157
- `ESP_GATT_UUID_CHAR_SVR_CONFIG` (C macro), 157
- `ESP_GATT_UUID_CHAR_VALID_RANGE` (C macro), 157
- `ESP_GATT_UUID_CONT_GLUCOSE_MONITOR_SVC` (C macro), 157
- `ESP_GATT_UUID_CSC_FEATURE` (C macro), 159
- `ESP_GATT_UUID_CSC_MEASUREMENT` (C macro), 159
- `ESP_GATT_UUID_CURRENT_TIME` (C macro), 158
- `ESP_GATT_UUID_CURRENT_TIME_SVC` (C macro), 156
- `ESP_GATT_UUID_CYCLING_POWER_SVC` (C macro), 157
- `ESP_GATT_UUID_CYCLING_SPEED_CADENCE_SVC` (C macro), 157
- `ESP_GATT_UUID_DEVICE_INFO_SVC` (C macro), 156
- `ESP_GATT_UUID_ENV_SENSING_CONFIG_DESCR` (C macro), 157
- `ESP_GATT_UUID_ENV_SENSING_MEASUREMENT_DESCR` (C macro), 157
- `ESP_GATT_UUID_ENV_SENSING_TRIGGER_DESCR` (C macro), 157
- `ESP_GATT_UUID_ENVIRONMENTAL_SENSING_SVC` (C macro), 157
- `ESP_GATT_UUID_EXT_RPT_REF_DESCR` (C macro), 157
- `ESP_GATT_UUID_FW_VERSION_STR` (C macro), 158
- `ESP_GATT_UUID_GAP_CENTRAL_ADDR_RESOL` (C macro), 157
- `ESP_GATT_UUID_GAP_DEVICE_NAME` (C macro), 157
- `ESP_GATT_UUID_GAP_ICON` (C macro), 157
- `ESP_GATT_UUID_GAP_PREF_CONN_PARAM` (C macro), 157
- `ESP_GATT_UUID_GATT_SRV_CHGD` (C macro), 157

- ESP_GATT_UUID_GLUCOSE_SVC (*C macro*), 156
- ESP_GATT_UUID_GM_CONTEXT (*C macro*), 158
- ESP_GATT_UUID_GM_CONTROL_POINT (*C macro*), 158
- ESP_GATT_UUID_GM_FEATURE (*C macro*), 158
- ESP_GATT_UUID_GM_MEASUREMENT (*C macro*), 158
- ESP_GATT_UUID_HEALTH_THERMOM_SVC (*C macro*), 156
- ESP_GATT_UUID_HEART_RATE_SVC (*C macro*), 156
- ESP_GATT_UUID_HID_BT_KB_INPUT (*C macro*), 158
- ESP_GATT_UUID_HID_BT_KB_OUTPUT (*C macro*), 158
- ESP_GATT_UUID_HID_BT_MOUSE_INPUT (*C macro*), 158
- ESP_GATT_UUID_HID_CONTROL_POINT (*C macro*), 158
- ESP_GATT_UUID_HID_INFORMATION (*C macro*), 158
- ESP_GATT_UUID_HID_PROTO_MODE (*C macro*), 158
- ESP_GATT_UUID_HID_REPORT (*C macro*), 158
- ESP_GATT_UUID_HID_REPORT_MAP (*C macro*), 158
- ESP_GATT_UUID_HID_SVC (*C macro*), 157
- ESP_GATT_UUID_HW_VERSION_STR (*C macro*), 158
- ESP_GATT_UUID_IEEE_DATA (*C macro*), 158
- ESP_GATT_UUID_IMMEDIATE_ALERT_SVC (*C macro*), 156
- ESP_GATT_UUID_INCLUDE_SERVICE (*C macro*), 157
- ESP_GATT_UUID_LINK_LOSS_SVC (*C macro*), 156
- ESP_GATT_UUID_LOCAL_TIME_INFO (*C macro*), 158
- ESP_GATT_UUID_LOCATION_AND_NAVIGATION_SVC (*C macro*), 157
- ESP_GATT_UUID_MANU_NAME (*C macro*), 158
- ESP_GATT_UUID_MODEL_NUMBER_STR (*C macro*), 158
- ESP_GATT_UUID_NEXT_DST_CHANGE_SVC (*C macro*), 156
- ESP_GATT_UUID_NUM_DIGITALS_DESCR (*C macro*), 157
- ESP_GATT_UUID_NW_STATUS (*C macro*), 158
- ESP_GATT_UUID_NW_TRIGGER (*C macro*), 158
- ESP_GATT_UUID_PHONE_ALERT_STATUS_SVC (*C macro*), 156
- ESP_GATT_UUID_PNP_ID (*C macro*), 158
- ESP_GATT_UUID_PRI_SERVICE (*C macro*), 157
- ESP_GATT_UUID_REF_TIME_INFO (*C macro*), 158
- ESP_GATT_UUID_REF_TIME_UPDATE_SVC (*C macro*), 156
- ESP_GATT_UUID_RINGER_CP (*C macro*), 158
- ESP_GATT_UUID_RINGER_SETTING (*C macro*), 158
- ESP_GATT_UUID_RPT_REF_DESCR (*C macro*), 157
- ESP_GATT_UUID_RSC_FEATURE (*C macro*), 158
- ESP_GATT_UUID_RSC_MEASUREMENT (*C macro*), 158
- ESP_GATT_UUID_RUNNING_SPEED_CADENCE_SVC (*C macro*), 157
- ESP_GATT_UUID_SC_CONTROL_POINT (*C macro*), 158
- ESP_GATT_UUID_SCAN_INT_WINDOW (*C macro*), 159
- ESP_GATT_UUID_SCAN_PARAMETERS_SVC (*C macro*), 157
- ESP_GATT_UUID_SCAN_REFRESH (*C macro*), 159
- ESP_GATT_UUID_SEC_SERVICE (*C macro*), 157
- ESP_GATT_UUID_SENSOR_LOCATION (*C macro*), 158
- ESP_GATT_UUID_SERIAL_NUMBER_STR (*C macro*), 158
- ESP_GATT_UUID_SW_VERSION_STR (*C macro*), 158
- ESP_GATT_UUID_SYSTEM_ID (*C macro*), 158
- ESP_GATT_UUID_TIME_TRIGGER_DESCR (*C macro*), 157
- ESP_GATT_UUID_TX_POWER_LEVEL (*C macro*), 157
- ESP_GATT_UUID_TX_POWER_SVC (*C macro*), 156
- ESP_GATT_UUID_USER_DATA_SVC (*C macro*), 157
- ESP_GATT_UUID_VALUE_TRIGGER_DESCR (*C macro*), 157
- ESP_GATT_UUID_WEIGHT_SCALE_SVC (*C macro*), 157
- esp_gatt_value_t (*C++ class*), 154
- esp_gatt_value_t::auth_req (*C++ member*), 154
- esp_gatt_value_t::handle (*C++ member*), 154
- esp_gatt_value_t::len (*C++ member*), 154
- esp_gatt_value_t::offset (*C++ member*), 154
- esp_gatt_value_t::value (*C++ member*), 154
- ESP_GATT_WRITE_NOT_PERMIT (*C++ enumerator*), 160
- ESP_GATT_WRITE_TYPE_NO_RSP (*C++ enumerator*), 162
- ESP_GATT_WRITE_TYPE_RSP (*C++ enumerator*), 162
- esp_gatt_write_type_t (*C++ enum*), 162
- ESP_GATT_WRONG_STATE (*C++ enumerator*), 160
- ESP_GATTC_ACL_EVT (*C++ enumerator*), 189
- ESP_GATTC_ADV_DATA_EVT (*C++ enumerator*), 189
- ESP_GATTC_ADV_VSC_EVT (*C++ enumerator*), 190
- ESP_GATTC_BTH_SCAN_CFG_EVT (*C++ enumerator*), 190

- ESP_GATTC_BTH_SCAN_DIS_EVT (C++ *enumerator*), 190
- ESP_GATTC_BTH_SCAN_ENB_EVT (C++ *enumerator*), 190
- ESP_GATTC_BTH_SCAN_PARAM_EVT (C++ *enumerator*), 190
- ESP_GATTC_BTH_SCAN_RD_EVT (C++ *enumerator*), 190
- ESP_GATTC_BTH_SCAN_THR_EVT (C++ *enumerator*), 190
- ESP_GATTC_CANCEL_OPEN_EVT (C++ *enumerator*), 189
- esp_gattc_cb_event_t (C++ *enum*), 188
- esp_gattc_cb_t (C++ *type*), 188
- ESP_GATTC_CFG_MTU_EVT (C++ *enumerator*), 189
- esp_gattc_char_elem_t (C++ *class*), 155
- esp_gattc_char_elem_t::char_handle (C++ *member*), 156
- esp_gattc_char_elem_t::properties (C++ *member*), 156
- esp_gattc_char_elem_t::uuid (C++ *member*), 156
- ESP_GATTC_CLOSE_EVT (C++ *enumerator*), 189
- ESP_GATTC_CONGEST_EVT (C++ *enumerator*), 189
- ESP_GATTC_CONNECT_EVT (C++ *enumerator*), 190
- esp_gattc_db_elem_t (C++ *class*), 155
- esp_gattc_db_elem_t::attribute_handle (C++ *member*), 155
- esp_gattc_db_elem_t::end_handle (C++ *member*), 155
- esp_gattc_db_elem_t::properties (C++ *member*), 155
- esp_gattc_db_elem_t::start_handle (C++ *member*), 155
- esp_gattc_db_elem_t::type (C++ *member*), 155
- esp_gattc_db_elem_t::uuid (C++ *member*), 155
- esp_gattc_descr_elem_t (C++ *class*), 156
- esp_gattc_descr_elem_t::handle (C++ *member*), 156
- esp_gattc_descr_elem_t::uuid (C++ *member*), 156
- ESP_GATTC_DIS_SRVC_CMPL_EVT (C++ *enumerator*), 190
- ESP_GATTC_DISCONNECT_EVT (C++ *enumerator*), 190
- ESP_GATTC_ENC_CMPL_CB_EVT (C++ *enumerator*), 189
- ESP_GATTC_EXEC_EVT (C++ *enumerator*), 189
- ESP_GATTC_GET_ADDR_LIST_EVT (C++ *enumerator*), 190
- esp_gattc_incl_svc_elem_t (C++ *class*), 156
- esp_gattc_incl_svc_elem_t::handle (C++ *member*), 156
- esp_gattc_incl_svc_elem_t::incl_srvc_elem_t (C++ *member*), 156
- esp_gattc_incl_svc_elem_t::incl_srvc_service_elem_t (C++ *member*), 156
- esp_gattc_multi_t (C++ *class*), 155
- esp_gattc_multi_t::handles (C++ *member*), 155
- esp_gattc_multi_t::num_attr (C++ *member*), 155
- ESP_GATTC_NOTIFY_EVT (C++ *enumerator*), 189
- ESP_GATTC_OPEN_EVT (C++ *enumerator*), 189
- ESP_GATTC_PREP_WRITE_EVT (C++ *enumerator*), 189
- ESP_GATTC_QUEUE_FULL_EVT (C++ *enumerator*), 190
- ESP_GATTC_READ_CHAR_EVT (C++ *enumerator*), 189
- ESP_GATTC_READ_DESCR_EVT (C++ *enumerator*), 189
- ESP_GATTC_READ_MULTIPLE_EVT (C++ *enumerator*), 190
- ESP_GATTC_REG_EVT (C++ *enumerator*), 188
- ESP_GATTC_REG_FOR_NOTIFY_EVT (C++ *enumerator*), 190
- ESP_GATTC_SCAN_FLT_CFG_EVT (C++ *enumerator*), 190
- ESP_GATTC_SCAN_FLT_PARAM_EVT (C++ *enumerator*), 190
- ESP_GATTC_SCAN_FLT_STATUS_EVT (C++ *enumerator*), 190
- ESP_GATTC_SEARCH_CMPL_EVT (C++ *enumerator*), 189
- ESP_GATTC_SEARCH_RES_EVT (C++ *enumerator*), 189
- esp_gattc_service_elem_t (C++ *class*), 155
- esp_gattc_service_elem_t::end_handle (C++ *member*), 155
- esp_gattc_service_elem_t::is_primary (C++ *member*), 155
- esp_gattc_service_elem_t::start_handle (C++ *member*), 155
- esp_gattc_service_elem_t::uuid (C++ *member*), 155
- ESP_GATTC_SET_ASSOC_EVT (C++ *enumerator*), 190
- ESP_GATTC_SRVC_CHG_EVT (C++ *enumerator*), 189
- ESP_GATTC_UNREG_EVT (C++ *enumerator*), 189
- ESP_GATTC_UNREG_FOR_NOTIFY_EVT (C++ *enumerator*), 190
- ESP_GATTC_WRITE_CHAR_EVT (C++ *enumerator*), 189

- 189
- ESP_GATTC_WRITE_DESCR_EVT (C++ *enumerator*), 189
- ESP_GATTS_ADD_CHAR_DESCR_EVT (C++ *enumerator*), 174
- ESP_GATTS_ADD_CHAR_EVT (C++ *enumerator*), 174
- ESP_GATTS_ADD_INCL_SRVC_EVT (C++ *enumerator*), 174
- esp_gatts_attr_db_t (C++ *class*), 153
- esp_gatts_attr_db_t::att_desc (C++ *member*), 153
- esp_gatts_attr_db_t::attr_control (C++ *member*), 153
- ESP_GATTS_CANCEL_OPEN_EVT (C++ *enumerator*), 174
- esp_gatts_cb_event_t (C++ *enum*), 173
- esp_gatts_cb_t (C++ *type*), 173
- ESP_GATTS_CLOSE_EVT (C++ *enumerator*), 174
- ESP_GATTS_CONF_EVT (C++ *enumerator*), 174
- ESP_GATTS_CONGEST_EVT (C++ *enumerator*), 174
- ESP_GATTS_CONNECT_EVT (C++ *enumerator*), 174
- ESP_GATTS_CREAT_ATTR_TAB_EVT (C++ *enumerator*), 174
- ESP_GATTS_CREATE_EVT (C++ *enumerator*), 174
- ESP_GATTS_DELETE_EVT (C++ *enumerator*), 174
- ESP_GATTS_DISCONNECT_EVT (C++ *enumerator*), 174
- ESP_GATTS_EXEC_WRITE_EVT (C++ *enumerator*), 173
- esp_gatts_incl128_svc_desc_t (C++ *class*), 154
- esp_gatts_incl128_svc_desc_t::end_hdl (C++ *member*), 154
- esp_gatts_incl128_svc_desc_t::start_hdl (C++ *member*), 154
- esp_gatts_incl_svc_desc_t (C++ *class*), 154
- esp_gatts_incl_svc_desc_t::end_hdl (C++ *member*), 154
- esp_gatts_incl_svc_desc_t::start_hdl (C++ *member*), 154
- esp_gatts_incl_svc_desc_t::uuid (C++ *member*), 154
- ESP_GATTS_LISTEN_EVT (C++ *enumerator*), 174
- ESP_GATTS_MTU_EVT (C++ *enumerator*), 173
- ESP_GATTS_OPEN_EVT (C++ *enumerator*), 174
- ESP_GATTS_READ_EVT (C++ *enumerator*), 173
- ESP_GATTS_REG_EVT (C++ *enumerator*), 173
- ESP_GATTS_RESPONSE_EVT (C++ *enumerator*), 174
- ESP_GATTS_SEND_SERVICE_CHANGE_EVT (C++ *enumerator*), 174
- ESP_GATTS_SET_ATTR_VAL_EVT (C++ *enumerator*), 174
- ESP_GATTS_START_EVT (C++ *enumerator*), 174
- ESP_GATTS_STOP_EVT (C++ *enumerator*), 174
- ESP_GATTS_UNREG_EVT (C++ *enumerator*), 174
- ESP_GATTS_WRITE_EVT (C++ *enumerator*), 173
- esp_gcov_dump (C++ *function*), 1006
- esp_get_deep_sleep_wake_stub (C++ *function*), 1202
- esp_get_flash_encryption_mode (C++ *function*), 981
- esp_get_free_heap_size (C++ *function*), 1174
- esp_get_free_internal_heap_size (C++ *function*), 1174
- esp_get_idf_version (C++ *function*), 1178
- esp_get_minimum_free_heap_size (C++ *function*), 1174
- esp_hf_answer_call_cmd_callback (C++ *type*), 246
- ESP_HF_AT_RESPONSE_CODE_BLACKLISTED (C++ *enumerator*), 252
- ESP_HF_AT_RESPONSE_CODE_BUSY (C++ *enumerator*), 252
- ESP_HF_AT_RESPONSE_CODE_CME (C++ *enumerator*), 252
- ESP_HF_AT_RESPONSE_CODE_DELAYED (C++ *enumerator*), 252
- ESP_HF_AT_RESPONSE_CODE_ERR (C++ *enumerator*), 251
- ESP_HF_AT_RESPONSE_CODE_NO_ANSWER (C++ *enumerator*), 252
- ESP_HF_AT_RESPONSE_CODE_NO_CARRIER (C++ *enumerator*), 252
- ESP_HF_AT_RESPONSE_CODE_OK (C++ *enumerator*), 251
- esp_hf_at_response_code_t (C++ *enum*), 251
- ESP_HF_AT_RESPONSE_ERROR (C++ *enumerator*), 252
- ESP_HF_AT_RESPONSE_OK (C++ *enumerator*), 252
- esp_hf_at_response_t (C++ *enum*), 252
- ESP_HF_ATA_RESPONSE_EVT (C++ *enumerator*), 274
- esp_hf_audio_state_callback (C++ *type*), 246
- ESP_HF_AUDIO_STATE_CONNECTED (C++ *enumerator*), 248
- ESP_HF_AUDIO_STATE_CONNECTED_MSBC (C++ *enumerator*), 248
- ESP_HF_AUDIO_STATE_CONNECTING (C++ *enumerator*), 248
- ESP_HF_AUDIO_STATE_DISCONNECTED (C++ *enumerator*), 248
- ESP_HF_AUDIO_STATE_EVT (C++ *enumerator*), 274
- esp_hf_audio_state_t (C++ *enum*), 248
- ESP_HF_BCS_RESPONSE_EVT (C++ *enumerator*), 274
- ESP_HF_BTRH_CMD_ACCEPT (C++ *enumerator*), 250
- ESP_HF_BTRH_CMD_HOLD (C++ *enumerator*), 250
- ESP_HF_BTRH_CMD_REJECT (C++ *enumerator*), 250
- esp_hf_btrh_cmd_t (C++ *enum*), 250
- ESP_HF_BTRH_STATUS_ACCEPTED (C++ *enumerator*), 250

- ator), 250
- ESP_HF_BTRH_STATUS_HELD (C++ enumerator), 250
- ESP_HF_BTRH_STATUS_REJECTED (C++ enumerator), 250
- esp_hf_btrh_status_t (C++ enum), 250
- ESP_HF_BVRA_RESPONSE_EVT (C++ enumerator), 274
- ESP_HF_CALL_ADDR_TYPE_INTERNATIONAL (C++ enumerator), 250
- esp_hf_call_addr_type_t (C++ enum), 250
- ESP_HF_CALL_ADDR_TYPE_UNKNOWN (C++ enumerator), 250
- ESP_HF_CALL_HELD_STATUS_HELD (C++ enumerator), 249
- ESP_HF_CALL_HELD_STATUS_HELD_AND_ACTIVE (C++ enumerator), 249
- ESP_HF_CALL_HELD_STATUS_NONE (C++ enumerator), 249
- esp_hf_call_held_status_t (C++ enum), 249
- ESP_HF_CALL_SETUP_STATUS_IDLE (C++ enumerator), 248
- ESP_HF_CALL_SETUP_STATUS_INCOMING (C++ enumerator), 248
- ESP_HF_CALL_SETUP_STATUS_OUTGOING_ALERTING (C++ enumerator), 248
- ESP_HF_CALL_SETUP_STATUS_OUTGOING_DIALING (C++ enumerator), 248
- esp_hf_call_setup_status_t (C++ enum), 248
- ESP_HF_CALL_STATUS_CALL_IN_PROGRESS (C++ enumerator), 248
- ESP_HF_CALL_STATUS_NO_CALLS (C++ enumerator), 248
- esp_hf_call_status_t (C++ enum), 248
- ESP_HF_CALL_WAITING_ACTIVE (C++ enumerator), 251
- ESP_HF_CALL_WAITING_INACTIVE (C++ enumerator), 251
- esp_hf_call_waiting_status_t (C++ enum), 251
- esp_hf_cb_event_t (C++ enum), 274
- esp_hf_cb_param_t (C++ union), 270
- esp_hf_cb_param_t::audio_stat (C++ member), 270
- esp_hf_cb_param_t::bcs_rep (C++ member), 270
- esp_hf_cb_param_t::cind (C++ member), 270
- esp_hf_cb_param_t::conn_stat (C++ member), 270
- esp_hf_cb_param_t::hf_audio_stat_param (C++ class), 270
- esp_hf_cb_param_t::hf_audio_stat_param::remote_addr (C++ member), 271
- esp_hf_cb_param_t::hf_audio_stat_param::state (C++ member), 271
- esp_hf_cb_param_t::hf_bcs_rep_param (C++ class), 271
- esp_hf_cb_param_t::hf_bcs_rep_param::mode (C++ member), 271
- esp_hf_cb_param_t::hf_cind_param (C++ class), 271
- esp_hf_cb_param_t::hf_cind_param::battery_level (C++ member), 271
- esp_hf_cb_param_t::hf_cind_param::call_held_status (C++ member), 271
- esp_hf_cb_param_t::hf_cind_param::call_setup_status (C++ member), 271
- esp_hf_cb_param_t::hf_cind_param::call_status (C++ member), 271
- esp_hf_cb_param_t::hf_cind_param::roam (C++ member), 271
- esp_hf_cb_param_t::hf_cind_param::signal_strength (C++ member), 271
- esp_hf_cb_param_t::hf_cind_param::svc (C++ member), 271
- esp_hf_cb_param_t::hf_conn_stat_param (C++ class), 271
- esp_hf_cb_param_t::hf_conn_stat_param::chld_feat (C++ member), 271
- esp_hf_cb_param_t::hf_conn_stat_param::peer_feat (C++ member), 271
- esp_hf_cb_param_t::hf_conn_stat_param::remote_bda (C++ member), 271
- esp_hf_cb_param_t::hf_conn_stat_param::state (C++ member), 271
- esp_hf_cb_param_t::hf_nrec_param (C++ class), 272
- esp_hf_cb_param_t::hf_nrec_param::state (C++ member), 272
- esp_hf_cb_param_t::hf_out_call_param (C++ class), 272
- esp_hf_cb_param_t::hf_out_call_param::num_or_loc (C++ member), 272
- esp_hf_cb_param_t::hf_out_call_param::remote_addr (C++ member), 272
- esp_hf_cb_param_t::hf_unat_rep_param (C++ class), 272
- esp_hf_cb_param_t::hf_unat_rep_param::unat (C++ member), 272
- esp_hf_cb_param_t::hf_volume_control_param (C++ class), 272
- esp_hf_cb_param_t::hf_volume_control_param::type (C++ member), 272
- esp_hf_cb_param_t::hf_volume_control_param::volume (C++ member), 272
- esp_hf_cb_param_t::hf_vra_rep_param (C++ class), 272
- esp_hf_cb_param_t::hf_vra_rep_param::remote_addr (C++ member), 272
- esp_hf_cb_param_t::hf_vra_rep_param::value (C++ member), 272
- esp_hf_cb_param_t::hf_vts_rep_param (C++ class), 272
- esp_hf_cb_param_t::hf_vts_rep_param::code (C++ member), 273

- esp_hf_cb_param_t::hf_wbs_rep_param (C++ class), 273
 esp_hf_cb_param_t::hf_wbs_rep_param::codec (C++ member), 273
 esp_hf_cb_param_t::nrec (C++ member), 270
 esp_hf_cb_param_t::out_call (C++ member), 270
 esp_hf_cb_param_t::unat_rep (C++ member), 270
 esp_hf_cb_param_t::volume_control (C++ member), 270
 esp_hf_cb_param_t::vra_rep (C++ member), 270
 esp_hf_cb_param_t::vts_rep (C++ member), 270
 esp_hf_cb_param_t::wbs_rep (C++ member), 270
 esp_hf_cb_t (C++ type), 274
 esp_hf_chld_cmd_callback (C++ type), 247
 ESP_HF_CHLD_FEAT_HOLD_ACC (C macro), 273
 ESP_HF_CHLD_FEAT_MERGE (C macro), 273
 ESP_HF_CHLD_FEAT_MERGE_DETACH (C macro), 273
 ESP_HF_CHLD_FEAT_PRIV_X (C macro), 273
 ESP_HF_CHLD_FEAT_REL (C macro), 273
 ESP_HF_CHLD_FEAT_REL_ACC (C macro), 273
 ESP_HF_CHLD_FEAT_REL_X (C macro), 273
 ESP_HF_CHLD_TYPE_HOLD_ACC (C++ enumerator), 251
 ESP_HF_CHLD_TYPE_MERGE (C++ enumerator), 251
 ESP_HF_CHLD_TYPE_MERGE_DETACH (C++ enumerator), 251
 ESP_HF_CHLD_TYPE_PRIV_X (C++ enumerator), 251
 ESP_HF_CHLD_TYPE_REL (C++ enumerator), 251
 ESP_HF_CHLD_TYPE_REL_ACC (C++ enumerator), 251
 ESP_HF_CHLD_TYPE_REL_X (C++ enumerator), 251
 esp_hf_chld_type_t (C++ enum), 251
 ESP_HF_CHUP_RESPONSE_EVT (C++ enumerator), 274
 esp_hf_cind_cmd_callback (C++ type), 247
 ESP_HF_CIND_RESPONSE_EVT (C++ enumerator), 274
 esp_hf_clcc_cmd_callback (C++ type), 247
 ESP_HF_CLCC_RESPONSE_EVT (C++ enumerator), 274
 esp_hf_client_answer_call (C++ function), 255
 ESP_HF_CLIENT_AT_RESPONSE_EVT (C++ enumerator), 264
 ESP_HF_CLIENT_AUDIO_STATE_CONNECTED (C++ enumerator), 263
 ESP_HF_CLIENT_AUDIO_STATE_CONNECTED_MSBC (C++ enumerator), 263
 ESP_HF_CLIENT_AUDIO_STATE_CONNECTING (C++ enumerator), 263
 ESP_HF_CLIENT_AUDIO_STATE_DISCONNECTED (C++ enumerator), 263
 ESP_HF_CLIENT_AUDIO_STATE_EVT (C++ enumerator), 264
 esp_hf_client_audio_state_t (C++ enum), 263
 ESP_HF_CLIENT_BINP_EVT (C++ enumerator), 264
 ESP_HF_CLIENT_BSIR_EVT (C++ enumerator), 264
 ESP_HF_CLIENT_BTRH_EVT (C++ enumerator), 264
 ESP_HF_CLIENT_BVRA_EVT (C++ enumerator), 264
 esp_hf_client_cb_event_t (C++ enum), 263
 esp_hf_client_cb_param_t (C++ union), 257
 esp_hf_client_cb_param_t::at_response (C++ member), 258
 esp_hf_client_cb_param_t::audio_stat (C++ member), 257
 esp_hf_client_cb_param_t::battery_level (C++ member), 258
 esp_hf_client_cb_param_t::binp (C++ member), 258
 esp_hf_client_cb_param_t::bsir (C++ member), 258
 esp_hf_client_cb_param_t::btrh (C++ member), 258
 esp_hf_client_cb_param_t::bvra (C++ member), 257
 esp_hf_client_cb_param_t::call (C++ member), 258
 esp_hf_client_cb_param_t::call_held (C++ member), 258
 esp_hf_client_cb_param_t::call_setup (C++ member), 258
 esp_hf_client_cb_param_t::ccwa (C++ member), 258
 esp_hf_client_cb_param_t::clcc (C++ member), 258
 esp_hf_client_cb_param_t::clip (C++ member), 258
 esp_hf_client_cb_param_t::cnum (C++ member), 258
 esp_hf_client_cb_param_t::conn_stat (C++ member), 257
 esp_hf_client_cb_param_t::cops (C++ member), 258
 esp_hf_client_cb_param_t::hf_client_at_response_p (C++ class), 258
 esp_hf_client_cb_param_t::hf_client_at_response_p (C++ member), 258
 esp_hf_client_cb_param_t::hf_client_at_response_p (C++ member), 258
 esp_hf_client_cb_param_t::hf_client_at_response_p (C++ member), 258
 esp_hf_client_cb_param_t::hf_client_audio_stat_pa (C++ class), 258
 esp_hf_client_cb_param_t::hf_client_audio_stat_pa (C++ member), 258

- ESP_HF_CLIENT_CHLD_FEAT_REL_X (C macro), 262
- ESP_HF_CLIENT_CIND_BATTERY_LEVEL_EVT (C++ enumerator), 264
- ESP_HF_CLIENT_CIND_CALL_EVT (C++ enumerator), 264
- ESP_HF_CLIENT_CIND_CALL_HELD_EVT (C++ enumerator), 264
- ESP_HF_CLIENT_CIND_CALL_SETUP_EVT (C++ enumerator), 264
- ESP_HF_CLIENT_CIND_ROAMING_STATUS_EVT (C++ enumerator), 264
- ESP_HF_CLIENT_CIND_SERVICE_AVAILABILITY_EVT (C++ enumerator), 264
- ESP_HF_CLIENT_CIND_SIGNAL_STRENGTH_EVT (C++ enumerator), 264
- ESP_HF_CLIENT_CLCC_EVT (C++ enumerator), 264
- ESP_HF_CLIENT_CLIP_EVT (C++ enumerator), 264
- ESP_HF_CLIENT_CNUM_EVT (C++ enumerator), 264
- esp_hf_client_connect (C++ function), 254
- esp_hf_client_connect_audio (C++ function), 254
- ESP_HF_CLIENT_CONNECTION_STATE_CONNECTED (C++ enumerator), 263
- ESP_HF_CLIENT_CONNECTION_STATE_CONNECTING (C++ enumerator), 263
- ESP_HF_CLIENT_CONNECTION_STATE_DISCONNECTED (C++ enumerator), 263
- ESP_HF_CLIENT_CONNECTION_STATE_DISCONNECTING (C++ enumerator), 263
- ESP_HF_CLIENT_CONNECTION_STATE_EVT (C++ enumerator), 264
- ESP_HF_CLIENT_CONNECTION_STATE_SLC_CONNECTED (C++ enumerator), 263
- esp_hf_client_connection_state_t (C++ enum), 263
- ESP_HF_CLIENT_COPS_CURRENT_OPERATOR_EVT (C++ enumerator), 264
- esp_hf_client_deinit (C++ function), 253
- esp_hf_client_dial (C++ function), 255
- esp_hf_client_dial_memory (C++ function), 255
- esp_hf_client_disconnect (C++ function), 254
- esp_hf_client_disconnect_audio (C++ function), 254
- esp_hf_client_in_band_ring_state_t (C++ enum), 263
- ESP_HF_CLIENT_IN_BAND_RINGTONE_NOT_PROVIDED (C++ enumerator), 263
- ESP_HF_CLIENT_IN_BAND_RINGTONE_PROVIDED (C++ enumerator), 263
- esp_hf_client_incoming_data_cb_t (C++ type), 262
- esp_hf_client_init (C++ function), 253
- esp_hf_client_outgoing_data_cb_t (C++ type), 263
- esp_hf_client_outgoing_data_ready (C++ function), 257
- esp_hf_client_pcm_resample (C++ function), 257
- esp_hf_client_pcm_resample_deinit (C++ function), 257
- esp_hf_client_pcm_resample_init (C++ function), 257
- ESP_HF_CLIENT_PEER_FEAT_3WAY (C macro), 262
- ESP_HF_CLIENT_PEER_FEAT_CODEC (C macro), 262
- ESP_HF_CLIENT_PEER_FEAT_ECC (C macro), 262
- ESP_HF_CLIENT_PEER_FEAT_ECNR (C macro), 262
- ESP_HF_CLIENT_PEER_FEAT_ECS (C macro), 262
- ESP_HF_CLIENT_PEER_FEAT_EXTERR (C macro), 262
- ESP_HF_CLIENT_PEER_FEAT_INBAND (C macro), 262
- ESP_HF_CLIENT_PEER_FEAT_REJECT (C macro), 262
- ESP_HF_CLIENT_PEER_FEAT_VREC (C macro), 262
- ESP_HF_CLIENT_PEER_FEAT_VTAG (C macro), 262
- esp_hf_client_query_current_calls (C++ function), 256
- esp_hf_client_query_current_operator_name (C++ function), 256
- esp_hf_client_register_callback (C++ function), 253
- esp_hf_client_register_data_callback (C++ function), 257
- esp_hf_client_reject_call (C++ function), 256
- esp_hf_client_request_last_voice_tag_number (C++ function), 256
- esp_hf_client_retrieve_subscriber_info (C++ function), 256
- ESP_HF_CLIENT_RING_IND_EVT (C++ enumerator), 264
- esp_hf_client_send_btrh_cmd (C++ function), 255
- esp_hf_client_send_chld_cmd (C++ function), 255
- esp_hf_client_send_dtmf (C++ function), 256
- esp_hf_client_send_nrec (C++ function), 256
- esp_hf_client_start_voice_recognition (C++ function), 254
- esp_hf_client_stop_voice_recognition (C++ function), 254
- ESP_HF_CLIENT_VOLUME_CONTROL_EVT (C++ enumerator), 264

- `esp_hf_client_volume_update` (C++ function), 255
- `ESP_HF_CME_AG_FAILURE` (C++ enumerator), 252
- `ESP_HF_CME_DIAL_STRING_TOO_LONG` (C++ enumerator), 253
- `esp_hf_cme_err_t` (C++ enum), 252
- `ESP_HF_CME_INCORRECT_PASSWORD` (C++ enumerator), 252
- `ESP_HF_CME_INVALID_CHARACTERS_IN_DIAL_STRING` (C++ enumerator), 253
- `ESP_HF_CME_INVALID_CHARACTERS_IN_TEXT_STRING` (C++ enumerator), 253
- `ESP_HF_CME_INVALID_INDEX` (C++ enumerator), 253
- `ESP_HF_CME_MEMEORY_FAILURE` (C++ enumerator), 253
- `ESP_HF_CME_MEMEORY_FULL` (C++ enumerator), 253
- `ESP_HF_CME_NETWORK_NOT_ALLOWED` (C++ enumerator), 253
- `ESP_HF_CME_NETWORK_TIMEOUT` (C++ enumerator), 253
- `ESP_HF_CME_NO_CONNECTION_TO_PHONE` (C++ enumerator), 252
- `ESP_HF_CME_NO_NETWORK_SERVICE` (C++ enumerator), 253
- `ESP_HF_CME_OPERATION_NOT_ALLOWED` (C++ enumerator), 252
- `ESP_HF_CME_OPERATION_NOT_SUPPORTED` (C++ enumerator), 252
- `ESP_HF_CME_PH_SIM_PIN_REQUIRED` (C++ enumerator), 252
- `ESP_HF_CME_SIM_BUSY` (C++ enumerator), 252
- `ESP_HF_CME_SIM_FAILURE` (C++ enumerator), 252
- `ESP_HF_CME_SIM_NOT_INSERTED` (C++ enumerator), 252
- `ESP_HF_CME_SIM_PIN2_REQUIRED` (C++ enumerator), 252
- `ESP_HF_CME_SIM_PIN_REQUIRED` (C++ enumerator), 252
- `ESP_HF_CME_SIM_PUK2_REQUIRED` (C++ enumerator), 252
- `ESP_HF_CME_SIM_PUK_REQUIRED` (C++ enumerator), 252
- `ESP_HF_CME_TEXT_STRING_TOO_LONG` (C++ enumerator), 253
- `esp_hf_cnum_cmd_callback` (C++ type), 247
- `ESP_HF_CNUM_RESPONSE_EVT` (C++ enumerator), 274
- `esp_hf_connection_state_callback` (C++ type), 246
- `ESP_HF_CONNECTION_STATE_CONNECTED` (C++ enumerator), 251
- `ESP_HF_CONNECTION_STATE_CONNECTING` (C++ enumerator), 251
- `ESP_HF_CONNECTION_STATE_DISCONNECTED` (C++ enumerator), 251
- `ESP_HF_CONNECTION_STATE_DISCONNECTING` (C++ enumerator), 251
- `ESP_HF_CONNECTION_STATE_EVT` (C++ enumerator), 274
- `ESP_HF_CONNECTION_STATE_SLC_CONNECTED` (C++ enumerator), 251
- `esp_hf_connection_state_t` (C++ enum), 251
- `esp_hf_cops_cmd_callback` (C++ type), 247
- `ESP_HF_COPS_RESPONSE_EVT` (C++ enumerator), 274
- `ESP_HF_CURRENT_CALL_DIRECTION_INCOMING` (C++ enumerator), 249
- `ESP_HF_CURRENT_CALL_DIRECTION_OUTGOING` (C++ enumerator), 249
- `esp_hf_current_call_direction_t` (C++ enum), 249
- `ESP_HF_CURRENT_CALL_MODE_DATA` (C++ enumerator), 250
- `ESP_HF_CURRENT_CALL_MODE_FAX` (C++ enumerator), 250
- `esp_hf_current_call_mode_t` (C++ enum), 250
- `ESP_HF_CURRENT_CALL_MODE_VOICE` (C++ enumerator), 250
- `ESP_HF_CURRENT_CALL_MPTY_TYPE_MULTI` (C++ enumerator), 249
- `ESP_HF_CURRENT_CALL_MPTY_TYPE_SINGLE` (C++ enumerator), 249
- `esp_hf_current_call_mpty_type_t` (C++ enum), 249
- `ESP_HF_CURRENT_CALL_STATUS_ACTIVE` (C++ enumerator), 249
- `ESP_HF_CURRENT_CALL_STATUS_ALERTING` (C++ enumerator), 249
- `ESP_HF_CURRENT_CALL_STATUS_DIALING` (C++ enumerator), 249
- `ESP_HF_CURRENT_CALL_STATUS_HELD` (C++ enumerator), 249
- `ESP_HF_CURRENT_CALL_STATUS_HELD_BY_RESP_HOLD` (C++ enumerator), 249
- `ESP_HF_CURRENT_CALL_STATUS_INCOMING` (C++ enumerator), 249
- `esp_hf_current_call_status_t` (C++ enum), 249
- `ESP_HF_CURRENT_CALL_STATUS_WAITING` (C++ enumerator), 249
- `esp_hf_dial_call_cmd_callback` (C++ type), 247
- `ESP_HF_DIAL_EVT` (C++ enumerator), 274
- `esp_hf_dtmf_cmd_callback` (C++ type), 247
- `esp_hf_hangup_call_cmd_callback` (C++ type), 246
- `esp_hf_in_band_ring_state_t` (C++ enum), 247
- `ESP_HF_IN_BAND_RINGTONE_NOT_PROVIDED` (C++ enumerator), 247
- `ESP_HF_IN_BAND_RINGTONE_PROVIDED` (C++ enumerator), 247

- `esp_hf_incoming_data_cb_t` (C++ type), 273
`ESP_HF_IND_UPDATE_EVT` (C++ enumerator), 274
`esp_hf_key_pressed_cmd_callback` (C++ type), 247
`ESP_HF_NETWORK_STATE_AVAILABLE` (C++ enumerator), 248
`ESP_HF_NETWORK_STATE_NOT_AVAILABLE` (C++ enumerator), 248
`esp_hf_network_state_t` (C++ enum), 248
`esp_hf_nrec_cmd_callback` (C++ type), 247
`ESP_HF_NREC_RESPONSE_EVT` (C++ enumerator), 274
`ESP_HF_NREC_START` (C++ enumerator), 250
`ESP_HF_NREC_STOP` (C++ enumerator), 250
`esp_hf_nrec_t` (C++ enum), 250
`esp_hf_outgoing_data_cb_t` (C++ type), 273
`esp_hf_outgoing_data_ready` (C++ function), 270
`ESP_HF_PEER_FEAT_3WAY` (C macro), 273
`ESP_HF_PEER_FEAT_CODEC` (C macro), 273
`ESP_HF_PEER_FEAT_ECC` (C macro), 273
`ESP_HF_PEER_FEAT_ECNR` (C macro), 273
`ESP_HF_PEER_FEAT_ECS` (C macro), 273
`ESP_HF_PEER_FEAT_EXTERR` (C macro), 273
`ESP_HF_PEER_FEAT_INBAND` (C macro), 273
`ESP_HF_PEER_FEAT_REJECT` (C macro), 273
`ESP_HF_PEER_FEAT_VREC` (C macro), 273
`ESP_HF_PEER_FEAT_VTAG` (C macro), 273
`ESP_HF_ROAMING_STATUS_ACTIVE` (C++ enumerator), 249
`ESP_HF_ROAMING_STATUS_INACTIVE` (C++ enumerator), 249
`esp_hf_roaming_status_t` (C++ enum), 249
`ESP_HF_SERVICE_TYPE_HOME` (C++ enumerator), 248
`ESP_HF_SERVICE_TYPE_ROAMING` (C++ enumerator), 248
`esp_hf_service_type_t` (C++ enum), 248
`ESP_HF_SUBSCRIBER_SERVICE_TYPE_FAX` (C++ enumerator), 250
`esp_hf_subscriber_service_type_t` (C++ enum), 250
`ESP_HF_SUBSCRIBER_SERVICE_TYPE_UNKNOWN` (C++ enumerator), 250
`ESP_HF_SUBSCRIBER_SERVICE_TYPE_VOICE` (C++ enumerator), 250
`esp_hf_unat_response` (C++ function), 266
`ESP_HF_UNAT_RESPONSE_EVT` (C++ enumerator), 274
`esp_hf_unknown_at_cmd_callback` (C++ type), 247
`esp_hf_volume_cmd_callback` (C++ type), 247
`ESP_HF_VOLUME_CONTROL_EVT` (C++ enumerator), 274
`ESP_HF_VOLUME_CONTROL_TARGET_MIC` (C++ enumerator), 248
`ESP_HF_VOLUME_CONTROL_TARGET_SPK` (C++ enumerator), 247
`esp_hf_volume_control_target_t` (C++ enum), 247
`ESP_HF_VOLUME_TYPE_MIC` (C++ enumerator), 248
`ESP_HF_VOLUME_TYPE_SPK` (C++ enumerator), 248
`esp_hf_volume_type_t` (C++ enum), 248
`esp_hf_vr_cmd_callback` (C++ type), 246
`ESP_HF_VR_STATE_DISABLED` (C++ enumerator), 247
`ESP_HF_VR_STATE_ENABLED` (C++ enumerator), 247
`esp_hf_vr_state_t` (C++ enum), 247
`ESP_HF_VTS_RESPONSE_EVT` (C++ enumerator), 274
`esp_hf_wbs_callback` (C++ type), 247
`esp_hf_wbs_config_t` (C++ enum), 251
`ESP_HF_WBS_NO` (C++ enumerator), 251
`ESP_HF_WBS_NONE` (C++ enumerator), 251
`ESP_HF_WBS_RESPONSE_EVT` (C++ enumerator), 274
`ESP_HF_WBS_YES` (C++ enumerator), 251
`esp_himem_alloc` (C++ function), 1157
`esp_himem_alloc_map_range` (C++ function), 1157
`ESP_HIMEM_BLKSZ` (C macro), 1158
`esp_himem_free` (C++ function), 1157
`esp_himem_free_map_range` (C++ function), 1157
`esp_himem_get_free_size` (C++ function), 1158
`esp_himem_get_phys_size` (C++ function), 1158
`esp_himem_handle_t` (C++ type), 1158
`esp_himem_map` (C++ function), 1157
`ESP_HIMEM_MAPFLAG_RO` (C macro), 1158
`esp_himem_rangehandle_t` (C++ type), 1158
`esp_himem_reserved_area_size` (C++ function), 1158
`esp_himem_unmap` (C++ function), 1158
`esp_http_client_add_auth` (C++ function), 827
`esp_http_client_auth_type_t` (C++ enum), 831
`esp_http_client_cleanup` (C++ function), 827
`esp_http_client_close` (C++ function), 827
`esp_http_client_config_t` (C++ class), 828
`esp_http_client_config_t::auth_type` (C++ member), 828
`esp_http_client_config_t::buffer_size` (C++ member), 829
`esp_http_client_config_t::buffer_size_tx` (C++ member), 829
`esp_http_client_config_t::cert_pem` (C++ member), 828
`esp_http_client_config_t::client_cert_pem` (C++ member), 828
`esp_http_client_config_t::client_key_pem`

- (C++ member), 829
- `esp_http_client_config_t::disable_auto_redirect` (C++ member), 829
- `esp_http_client_config_t::event_handler` (C++ member), 829
- `esp_http_client_config_t::host` (C++ member), 828
- `esp_http_client_config_t::is_async` (C++ member), 829
- `esp_http_client_config_t::keep_alive_count` (C++ member), 829
- `esp_http_client_config_t::keep_alive_enable` (C++ member), 829
- `esp_http_client_config_t::keep_alive_idle` (C++ member), 829
- `esp_http_client_config_t::keep_alive_interval` (C++ member), 829
- `esp_http_client_config_t::max_redirect_count` (C++ member), 829
- `esp_http_client_config_t::method` (C++ member), 829
- `esp_http_client_config_t::password` (C++ member), 828
- `esp_http_client_config_t::path` (C++ member), 828
- `esp_http_client_config_t::port` (C++ member), 828
- `esp_http_client_config_t::query` (C++ member), 828
- `esp_http_client_config_t::skip_cert_check` (C++ member), 829
- `esp_http_client_config_t::timeout_ms` (C++ member), 829
- `esp_http_client_config_t::transport_type` (C++ member), 829
- `esp_http_client_config_t::url` (C++ member), 828
- `esp_http_client_config_t::use_global_ca_cert` (C++ member), 829
- `esp_http_client_config_t::user_agent` (C++ member), 829
- `esp_http_client_config_t::user_data` (C++ member), 829
- `esp_http_client_config_t::username` (C++ member), 828
- `esp_http_client_delete_header` (C++ function), 825
- `esp_http_client_event` (C++ class), 828
- `esp_http_client_event::client` (C++ member), 828
- `esp_http_client_event::data` (C++ member), 828
- `esp_http_client_event::data_len` (C++ member), 828
- `esp_http_client_event::event_id` (C++ member), 828
- `esp_http_client_event::header_key` (C++ member), 828
- `esp_http_client_event::header_value` (C++ member), 828
- `esp_http_client_event::user_data` (C++ member), 828
- `esp_http_client_event_handle_t` (C++ type), 830
- `esp_http_client_event_id_t` (C++ enum), 830
- `esp_http_client_event_t` (C++ type), 830
- `esp_http_client_fetch_headers` (C++ function), 826
- `esp_http_client_get_content_length` (C++ function), 826
- `esp_http_client_get_header` (C++ function), 824
- `esp_http_client_get_password` (C++ function), 825
- `esp_http_client_get_post_field` (C++ function), 824
- `esp_http_client_get_status_code` (C++ function), 826
- `esp_http_client_get_transport_type` (C++ function), 827
- `esp_http_client_get_username` (C++ function), 824
- `esp_http_client_handle_t` (C++ type), 830
- `esp_http_client_init` (C++ function), 823
- `esp_http_client_is_chunked_response` (C++ function), 826
- `esp_http_client_is_name_complete_data_received` (C++ function), 827
- `esp_http_client_method_t` (C++ enum), 830
- `esp_http_client_open` (C++ function), 826
- `esp_http_client_perform` (C++ function), 823
- `esp_http_client_read` (C++ function), 826
- `esp_http_client_set_auth_type` (C++ function), 825
- `esp_http_client_set_header` (C++ function), 824
- `esp_http_client_set_method` (C++ function), 825
- `esp_http_client_set_password` (C++ function), 825
- `esp_http_client_set_post_field` (C++ function), 824
- `esp_http_client_set_redirection` (C++ function), 827
- `esp_http_client_set_url` (C++ function), 824
- `esp_http_client_set_username` (C++ function), 825
- `esp_http_client_transport_t` (C++ enum), 830
- `esp_http_client_write` (C++ function), 826
- `esp_https_ota` (C++ function), 1020
- `esp_https_ota_begin` (C++ function), 1020
- `esp_https_ota_config_t` (C++ class), 1021
- `esp_https_ota_config_t::http_config` (C++ member), 1021

- esp_https_ota_finish (C++ function), 1021
 esp_https_ota_get_image_len_read (C++ function), 1021
 esp_https_ota_get_img_desc (C++ function), 1021
 esp_https_ota_handle_t (C++ type), 1022
 esp_https_ota_is_complete_data_received (C++ function), 1021
 esp_https_ota_perform (C++ function), 1020
 ESP_IDF_VERSION (C macro), 1178
 ESP_IDF_VERSION_MAJOR (C macro), 1178
 ESP_IDF_VERSION_MINOR (C macro), 1178
 ESP_IDF_VERSION_PATCH (C macro), 1178
 ESP_IDF_VERSION_VAL (C macro), 1178
 ESP_IMAGE_FLASH_SIZE_16MB (C++ enumerator), 1002
 ESP_IMAGE_FLASH_SIZE_1MB (C++ enumerator), 1002
 ESP_IMAGE_FLASH_SIZE_2MB (C++ enumerator), 1002
 ESP_IMAGE_FLASH_SIZE_4MB (C++ enumerator), 1002
 ESP_IMAGE_FLASH_SIZE_8MB (C++ enumerator), 1002
 ESP_IMAGE_FLASH_SIZE_MAX (C++ enumerator), 1002
 esp_image_flash_size_t (C++ enum), 1002
 ESP_IMAGE_HEADER_MAGIC (C macro), 1001
 esp_image_header_t (C++ class), 1000
 esp_image_header_t::chip_id (C++ member), 1000
 esp_image_header_t::entry_addr (C++ member), 1000
 esp_image_header_t::hash_appended (C++ member), 1000
 esp_image_header_t::magic (C++ member), 1000
 esp_image_header_t::min_chip_rev (C++ member), 1000
 esp_image_header_t::reserved (C++ member), 1000
 esp_image_header_t::segment_count (C++ member), 1000
 esp_image_header_t::spi_mode (C++ member), 1000
 esp_image_header_t::spi_pin_drv (C++ member), 1000
 esp_image_header_t::spi_size (C++ member), 1000
 esp_image_header_t::spi_speed (C++ member), 1000
 esp_image_header_t::wp_pin (C++ member), 1000
 ESP_IMAGE_MAX_SEGMENTS (C macro), 1001
 esp_image_segment_header_t (C++ class), 1000
 esp_image_segment_header_t::data_len (C++ member), 1001
 esp_image_segment_header_t::load_addr (C++ member), 1001
 esp_image_spi_freq_t (C++ enum), 1002
 ESP_IMAGE_SPI_MODE_DIO (C++ enumerator), 1002
 ESP_IMAGE_SPI_MODE_DOUT (C++ enumerator), 1002
 ESP_IMAGE_SPI_MODE_FAST_READ (C++ enumerator), 1002
 ESP_IMAGE_SPI_MODE_QIO (C++ enumerator), 1002
 ESP_IMAGE_SPI_MODE_QOUT (C++ enumerator), 1002
 ESP_IMAGE_SPI_MODE_SLOW_READ (C++ enumerator), 1002
 esp_image_spi_mode_t (C++ enum), 1002
 ESP_IMAGE_SPI_SPEED_20M (C++ enumerator), 1002
 ESP_IMAGE_SPI_SPEED_26M (C++ enumerator), 1002
 ESP_IMAGE_SPI_SPEED_40M (C++ enumerator), 1002
 ESP_IMAGE_SPI_SPEED_80M (C++ enumerator), 1002
 esp_int_wdt_init (C++ function), 1205
 esp_intr_alloc (C++ function), 1163
 esp_intr_alloc_intrstatus (C++ function), 1163
 ESP_INTR_DISABLE (C macro), 1166
 esp_intr_disable (C++ function), 1164
 ESP_INTR_ENABLE (C macro), 1166
 esp_intr_enable (C++ function), 1165
 ESP_INTR_FLAG_EDGE (C macro), 1165
 ESP_INTR_FLAG_HIGH (C macro), 1165
 ESP_INTR_FLAG_INTRDISABLED (C macro), 1165
 ESP_INTR_FLAG_IRAM (C macro), 1165
 ESP_INTR_FLAG_LEVEL1 (C macro), 1165
 ESP_INTR_FLAG_LEVEL2 (C macro), 1165
 ESP_INTR_FLAG_LEVEL3 (C macro), 1165
 ESP_INTR_FLAG_LEVEL4 (C macro), 1165
 ESP_INTR_FLAG_LEVEL5 (C macro), 1165
 ESP_INTR_FLAG_LEVEL6 (C macro), 1165
 ESP_INTR_FLAG_LEVELMASK (C macro), 1166
 ESP_INTR_FLAG_LOWMED (C macro), 1165
 ESP_INTR_FLAG_NMI (C macro), 1165
 ESP_INTR_FLAG_SHARED (C macro), 1165
 esp_intr_free (C++ function), 1164
 esp_intr_get_cpu (C++ function), 1164
 esp_intr_get_intno (C++ function), 1164
 esp_intr_mark_shared (C++ function), 1163
 esp_intr_noniram_disable (C++ function), 1165
 esp_intr_noniram_enable (C++ function), 1165
 esp_intr_reserve (C++ function), 1163
 esp_intr_set_in_iram (C++ function), 1165
 ESP_IO_CAP_IN (C macro), 144
 ESP_IO_CAP_IO (C macro), 144

- ESP_IO_CAP_KBDISP (*C macro*), 144
- ESP_IO_CAP_NONE (*C macro*), 144
- ESP_IO_CAP_OUT (*C macro*), 144
- esp_ip4addr_aton (*C++ function*), 582
- esp_ip4addr_ntoa (*C++ function*), 582
- esp_ipc_call (*C++ function*), 1159
- esp_ipc_call_blocking (*C++ function*), 1159
- ESP_LE_AUTH_BOND (*C macro*), 143
- ESP_LE_AUTH_NO_BOND (*C macro*), 143
- ESP_LE_AUTH_REQ_BOND_MITM (*C macro*), 143
- ESP_LE_AUTH_REQ_MITM (*C macro*), 143
- ESP_LE_AUTH_REQ_SC_BOND (*C macro*), 143
- ESP_LE_AUTH_REQ_SC_MITM (*C macro*), 143
- ESP_LE_AUTH_REQ_SC_MITM_BOND (*C macro*), 143
- ESP_LE_AUTH_REQ_SC_ONLY (*C macro*), 143
- ESP_LE_KEY_LCSRK (*C macro*), 143
- ESP_LE_KEY_LENC (*C macro*), 143
- ESP_LE_KEY_LID (*C macro*), 143
- ESP_LE_KEY_LLK (*C macro*), 143
- ESP_LE_KEY_NONE (*C macro*), 143
- ESP_LE_KEY_PCSRK (*C macro*), 143
- ESP_LE_KEY_PENC (*C macro*), 143
- ESP_LE_KEY_PID (*C macro*), 143
- ESP_LE_KEY_PLK (*C macro*), 143
- esp_light_sleep_start (*C++ function*), 1201
- ESP_LINE_ENDINGS_CR (*C++ enumerator*), 994
- ESP_LINE_ENDINGS_CRLF (*C++ enumerator*), 994
- ESP_LINE_ENDINGS_LF (*C++ enumerator*), 994
- esp_line_endings_t (*C++ enum*), 994
- esp_link_key (*C++ type*), 122
- esp_local_ctrl_add_property (*C++ function*), 862
- esp_local_ctrl_config (*C++ class*), 865
- esp_local_ctrl_config::handlers (*C++ member*), 865
- esp_local_ctrl_config::max_properties (*C++ member*), 865
- esp_local_ctrl_config::transport (*C++ member*), 865
- esp_local_ctrl_config::transport_config (*C++ member*), 865
- esp_local_ctrl_config_t (*C++ type*), 866
- esp_local_ctrl_get_property (*C++ function*), 863
- esp_local_ctrl_get_transport_ble (*C++ function*), 862
- esp_local_ctrl_get_transport_httpd (*C++ function*), 862
- esp_local_ctrl_handlers (*C++ class*), 864
- esp_local_ctrl_handlers::get_prop_value (*C++ member*), 864
- esp_local_ctrl_handlers::set_prop_values (*C++ member*), 865
- esp_local_ctrl_handlers::usr_ctx (*C++ member*), 865
- esp_local_ctrl_handlers::usr_ctx_free_fn (*C++ member*), 865
- esp_local_ctrl_handlers_t (*C++ type*), 866
- esp_local_ctrl_prop (*C++ class*), 863
- esp_local_ctrl_prop::ctx (*C++ member*), 864
- esp_local_ctrl_prop::ctx_free_fn (*C++ member*), 864
- esp_local_ctrl_prop::flags (*C++ member*), 864
- esp_local_ctrl_prop::name (*C++ member*), 863
- esp_local_ctrl_prop::size (*C++ member*), 864
- esp_local_ctrl_prop::type (*C++ member*), 863
- esp_local_ctrl_prop_t (*C++ type*), 865
- esp_local_ctrl_prop_val (*C++ class*), 864
- esp_local_ctrl_prop_val::data (*C++ member*), 864
- esp_local_ctrl_prop_val::free_fn (*C++ member*), 864
- esp_local_ctrl_prop_val::size (*C++ member*), 864
- esp_local_ctrl_prop_val_t (*C++ type*), 866
- esp_local_ctrl_remove_property (*C++ function*), 862
- esp_local_ctrl_set_handler (*C++ function*), 863
- esp_local_ctrl_start (*C++ function*), 862
- esp_local_ctrl_stop (*C++ function*), 862
- ESP_LOCAL_CTRL_TRANSPORT_BLE (*C macro*), 865
- esp_local_ctrl_transport_config_ble_t (*C++ type*), 866
- esp_local_ctrl_transport_config_httpd_t (*C++ type*), 866
- esp_local_ctrl_transport_config_t (*C++ union*), 863
- esp_local_ctrl_transport_config_t::ble (*C++ member*), 863
- esp_local_ctrl_transport_config_t::httpd (*C++ member*), 863
- ESP_LOCAL_CTRL_TRANSPORT_HTTPD (*C macro*), 865
- esp_local_ctrl_transport_t (*C++ type*), 866
- ESP_LOG_BUFFER_CHAR (*C macro*), 1169
- ESP_LOG_BUFFER_CHAR_LEVEL (*C macro*), 1169
- ESP_LOG_BUFFER_HEX (*C macro*), 1169
- ESP_LOG_BUFFER_HEX_LEVEL (*C macro*), 1169
- ESP_LOG_BUFFER_HEXDUMP (*C macro*), 1169
- ESP_LOG_DEBUG (*C++ enumerator*), 1171
- ESP_LOG_EARLY_IMPL (*C macro*), 1170
- esp_log_early_timestamp (*C++ function*), 1168
- ESP_LOG_ERROR (*C++ enumerator*), 1170
- ESP_LOG_INFO (*C++ enumerator*), 1171
- ESP_LOG_LEVEL_LOCAL (*C macro*), 1170
- esp_log_level_set (*C++ function*), 1168
- esp_log_level_t (*C++ enum*), 1170

- ESP_LOG_NONE (C++ enumerator), 1170
- esp_log_set_vprintf (C++ function), 1168
- esp_log_system_timestamp (C++ function), 1168
- esp_log_timestamp (C++ function), 1168
- ESP_LOG_VERBOSE (C++ enumerator), 1171
- ESP_LOG_WARN (C++ enumerator), 1171
- esp_log_write (C++ function), 1168
- esp_log_writev (C++ function), 1169
- ESP_LOGD (C macro), 1170
- ESP_LOGE (C macro), 1170
- ESP_LOGI (C macro), 1170
- ESP_LOGV (C macro), 1170
- ESP_LOGW (C macro), 1170
- ESP_MAC_BT (C++ enumerator), 1177
- ESP_MAC_ETH (C++ enumerator), 1177
- esp_mac_type_t (C++ enum), 1177
- ESP_MAC_WIFI_SOFTAP (C++ enumerator), 1177
- ESP_MAC_WIFI_STA (C++ enumerator), 1177
- esp_mesh_allow_root_conflicts (C++ function), 544
- esp_mesh_available_txupQ_num (C++ function), 543
- esp_mesh_connect (C++ function), 547
- esp_mesh_deinit (C++ function), 536
- esp_mesh_delete_group_id (C++ function), 544
- esp_mesh_disconnect (C++ function), 547
- esp_mesh_fix_root (C++ function), 545
- esp_mesh_flush_scan_result (C++ function), 547
- esp_mesh_flush_upstream_packets (C++ function), 547
- esp_mesh_get_ap_assoc_expire (C++ function), 543
- esp_mesh_get_ap_authmode (C++ function), 541
- esp_mesh_get_ap_connections (C++ function), 541
- esp_mesh_get_capacity_num (C++ function), 545
- esp_mesh_get_config (C++ function), 539
- esp_mesh_get_group_list (C++ function), 544
- esp_mesh_get_group_num (C++ function), 544
- esp_mesh_get_id (C++ function), 540
- esp_mesh_get_ie_crypto_key (C++ function), 545
- esp_mesh_get_layer (C++ function), 541
- esp_mesh_get_max_layer (C++ function), 540
- esp_mesh_get_parent_bssid (C++ function), 541
- esp_mesh_get_root_healing_delay (C++ function), 545
- esp_mesh_get_router (C++ function), 539
- esp_mesh_get_router_bssid (C++ function), 548
- esp_mesh_get_routing_table (C++ function), 543
- esp_mesh_get_routing_table_size (C++ function), 543
- esp_mesh_get_rx_pending (C++ function), 543
- esp_mesh_get_self_organized (C++ function), 542
- esp_mesh_get_subnet_nodes_list (C++ function), 547
- esp_mesh_get_subnet_nodes_num (C++ function), 547
- esp_mesh_get_total_node_num (C++ function), 543
- esp_mesh_get_tsf_time (C++ function), 548
- esp_mesh_get_tx_pending (C++ function), 543
- esp_mesh_get_type (C++ function), 540
- esp_mesh_get_vote_percentage (C++ function), 542
- esp_mesh_get_xon_qsize (C++ function), 544
- esp_mesh_init (C++ function), 536
- esp_mesh_is_my_group (C++ function), 545
- esp_mesh_is_root (C++ function), 541
- esp_mesh_is_root_conflicts_allowed (C++ function), 544
- esp_mesh_is_root_fixed (C++ function), 546
- esp_mesh_post_toDS_state (C++ function), 543
- esp_mesh_recv (C++ function), 537
- esp_mesh_recv_toDS (C++ function), 538
- esp_mesh_scan_get_ap_ie_len (C++ function), 546
- esp_mesh_scan_get_ap_record (C++ function), 546
- esp_mesh_send (C++ function), 536
- esp_mesh_send_block_time (C++ function), 537
- esp_mesh_set_ap_assoc_expire (C++ function), 542
- esp_mesh_set_ap_authmode (C++ function), 541
- esp_mesh_set_ap_connections (C++ function), 541
- esp_mesh_set_ap_password (C++ function), 540
- esp_mesh_set_capacity_num (C++ function), 545
- esp_mesh_set_config (C++ function), 539
- esp_mesh_set_group_id (C++ function), 544
- esp_mesh_set_id (C++ function), 539
- esp_mesh_set_ie_crypto_funcs (C++ function), 545
- esp_mesh_set_ie_crypto_key (C++ function), 545
- esp_mesh_set_max_layer (C++ function), 540
- esp_mesh_set_parent (C++ function), 546
- esp_mesh_set_root_healing_delay (C++ function), 545
- esp_mesh_set_router (C++ function), 539
- esp_mesh_set_self_organized (C++ function), 541

- esp_mesh_set_type (C++ function), 540
 esp_mesh_set_vote_percentage (C++ function), 542
 esp_mesh_set_xon_qsize (C++ function), 544
 esp_mesh_start (C++ function), 536
 esp_mesh_stop (C++ function), 536
 esp_mesh_switch_channel (C++ function), 547
 esp_mesh_waive_root (C++ function), 542
 esp_mqtt_client_config_t (C++ class), 806
 esp_mqtt_client_config_t::alpn_protos (C++ member), 808
 esp_mqtt_client_config_t::buffer_size (C++ member), 807
 esp_mqtt_client_config_t::cert_len (C++ member), 807
 esp_mqtt_client_config_t::cert_pem (C++ member), 807
 esp_mqtt_client_config_t::client_cert_len (C++ member), 808
 esp_mqtt_client_config_t::client_cert_pem (C++ member), 808
 esp_mqtt_client_config_t::client_id (C++ member), 807
 esp_mqtt_client_config_t::client_key_len (C++ member), 808
 esp_mqtt_client_config_t::client_key_pem (C++ member), 808
 esp_mqtt_client_config_t::clientkey_password (C++ member), 808
 esp_mqtt_client_config_t::clientkey_password_len (C++ member), 808
 esp_mqtt_client_config_t::disable_auto_reconnect (C++ member), 807
 esp_mqtt_client_config_t::disable_clean_session (C++ member), 807
 esp_mqtt_client_config_t::disable_keepalive (C++ member), 808
 esp_mqtt_client_config_t::ds_data (C++ member), 808
 esp_mqtt_client_config_t::event_handle (C++ member), 807
 esp_mqtt_client_config_t::event_loop_handle (C++ member), 807
 esp_mqtt_client_config_t::host (C++ member), 807
 esp_mqtt_client_config_t::keepalive (C++ member), 807
 esp_mqtt_client_config_t::lwt_msg (C++ member), 807
 esp_mqtt_client_config_t::lwt_msg_len (C++ member), 807
 esp_mqtt_client_config_t::lwt_qos (C++ member), 807
 esp_mqtt_client_config_t::lwt_retain (C++ member), 807
 esp_mqtt_client_config_t::lwt_topic (C++ member), 807
 esp_mqtt_client_config_t::network_timeout_ms (C++ member), 806
 (C++ member), 808
 esp_mqtt_client_config_t::out_buffer_size (C++ member), 808
 esp_mqtt_client_config_t::password (C++ member), 807
 esp_mqtt_client_config_t::port (C++ member), 807
 esp_mqtt_client_config_t::protocol_ver (C++ member), 808
 esp_mqtt_client_config_t::psk_hint_key (C++ member), 808
 esp_mqtt_client_config_t::reconnect_timeout_ms (C++ member), 808
 esp_mqtt_client_config_t::refresh_connection_after_disconnect (C++ member), 808
 esp_mqtt_client_config_t::skip_cert_common_name_check (C++ member), 808
 esp_mqtt_client_config_t::task_prio (C++ member), 807
 esp_mqtt_client_config_t::task_stack (C++ member), 807
 esp_mqtt_client_config_t::transport (C++ member), 808
 esp_mqtt_client_config_t::uri (C++ member), 807
 esp_mqtt_client_config_t::use_global_ca_store (C++ member), 808
 esp_mqtt_client_config_t::use_secure_element (C++ member), 808
 esp_mqtt_client_config_t::user_context (C++ member), 807
 esp_mqtt_client_config_t::username (C++ member), 807
 esp_mqtt_client_destroy (C++ function), 805
 esp_mqtt_client_disconnect (C++ function), 803
 esp_mqtt_client_enqueue (C++ function), 804
 esp_mqtt_client_get_outbox_size (C++ function), 805
 esp_mqtt_client_handle_t (C++ type), 809
 esp_mqtt_client_init (C++ function), 803
 esp_mqtt_client_publish (C++ function), 804
 esp_mqtt_client_reconnect (C++ function), 803
 esp_mqtt_client_register_event (C++ function), 805
 esp_mqtt_client_set_uri (C++ function), 803
 esp_mqtt_client_start (C++ function), 803
 esp_mqtt_client_stop (C++ function), 803
 esp_mqtt_client_subscribe (C++ function), 804
 esp_mqtt_client_unsubscribe (C++ function), 804
 esp_mqtt_connect_return_code_t (C++ enum), 810
 esp_mqtt_error_codes (C++ class), 805
 esp_mqtt_error_codes::connect_return_code

- [esp_mqtt_error_codes::error_type \(C++ member\), 806](#)
[esp_mqtt_error_codes::esp_tls_cert_verify_file_t \(C++ member\), 806](#)
[esp_mqtt_error_codes::esp_tls_last_esp_err \(C++ member\), 806](#)
[esp_mqtt_error_codes::esp_tls_stack_err \(C++ member\), 806](#)
[esp_mqtt_error_codes::esp_transport_socket_err_t \(C++ member\), 806](#)
[esp_mqtt_error_codes_t \(C++ type\), 809](#)
[esp_mqtt_error_type_t \(C++ enum\), 810](#)
[esp_mqtt_event_handle_t \(C++ type\), 809](#)
[esp_mqtt_event_id_t \(C++ enum\), 809](#)
[esp_mqtt_event_t \(C++ class\), 806](#)
[esp_mqtt_event_t::client \(C++ member\), 806](#)
[esp_mqtt_event_t::current_data_offset \(C++ member\), 806](#)
[esp_mqtt_event_t::data \(C++ member\), 806](#)
[esp_mqtt_event_t::data_len \(C++ member\), 806](#)
[esp_mqtt_event_t::error_handle \(C++ member\), 806](#)
[esp_mqtt_event_t::event_id \(C++ member\), 806](#)
[esp_mqtt_event_t::msg_id \(C++ member\), 806](#)
[esp_mqtt_event_t::session_present \(C++ member\), 806](#)
[esp_mqtt_event_t::topic \(C++ member\), 806](#)
[esp_mqtt_event_t::topic_len \(C++ member\), 806](#)
[esp_mqtt_event_t::total_data_len \(C++ member\), 806](#)
[esp_mqtt_event_t::user_context \(C++ member\), 806](#)
[esp_mqtt_protocol_ver_t \(C++ enum\), 811](#)
[esp_mqtt_set_config \(C++ function\), 805](#)
[esp_mqtt_transport_t \(C++ enum\), 810](#)
[esp_netif_action_connected \(C++ function\), 577](#)
[esp_netif_action_disconnected \(C++ function\), 577](#)
[esp_netif_action_got_ip \(C++ function\), 577](#)
[esp_netif_action_start \(C++ function\), 576](#)
[esp_netif_action_stop \(C++ function\), 577](#)
[esp_netif_attach \(C++ function\), 576](#)
[esp_netif_attach_wifi_ap \(C++ function\), 583](#)
[esp_netif_attach_wifi_station \(C++ function\), 583](#)
[esp_netif_create_default_wifi_ap \(C++ function\), 584](#)
[esp_netif_create_default_wifi_mesh_network \(C++ function\), 584](#)
[esp_netif_create_default_wifi_sta \(C++ function\), 584](#)
[esp_netif_create_ip6_linklocal \(C++ function\), 581](#)
[esp_netif_deinit \(C++ function\), 575](#)
[esp_netif_destroy \(C++ function\), 576](#)
[esp_netif_dhcpc_get_status \(C++ function\), 580](#)
[esp_netif_dhcpc_option \(C++ function\), 579](#)
[esp_netif_dhcpc_start \(C++ function\), 580](#)
[esp_netif_dhcpc_stop \(C++ function\), 580](#)
[esp_netif_dhcps_get_status \(C++ function\), 580](#)
[esp_netif_dhcps_option \(C++ function\), 579](#)
[esp_netif_dhcps_start \(C++ function\), 580](#)
[esp_netif_dhcps_stop \(C++ function\), 580](#)
[esp_netif_free_rx_buffer \(C++ function\), 587](#)
[esp_netif_get_desc \(C++ function\), 583](#)
[esp_netif_get_dns_info \(C++ function\), 581](#)
[esp_netif_get_event_id \(C++ function\), 583](#)
[esp_netif_get_flags \(C++ function\), 583](#)
[esp_netif_get_handle_from_ifkey \(C++ function\), 582](#)
[esp_netif_get_handle_from_netif_impl \(C++ function\), 587](#)
[esp_netif_get_hostname \(C++ function\), 578](#)
[esp_netif_get_ifkey \(C++ function\), 583](#)
[esp_netif_get_io_driver \(C++ function\), 582](#)
[esp_netif_get_ip6_global \(C++ function\), 582](#)
[esp_netif_get_ip6_linklocal \(C++ function\), 581](#)
[esp_netif_get_ip_info \(C++ function\), 578](#)
[esp_netif_get_netif_impl \(C++ function\), 587](#)
[esp_netif_get_netif_impl_index \(C++ function\), 579](#)
[esp_netif_get_nr_of_ifs \(C++ function\), 583](#)
[esp_netif_get_old_ip_info \(C++ function\), 578](#)
[esp_netif_init \(C++ function\), 575](#)
[esp_netif_is_netif_up \(C++ function\), 578](#)
[esp_netif_new \(C++ function\), 576](#)
[esp_netif_next \(C++ function\), 583](#)
[esp_netif_receive \(C++ function\), 576](#)
[esp_netif_set_dns_info \(C++ function\), 581](#)
[esp_netif_set_driver_config \(C++ function\), 576](#)
[esp_netif_set_hostname \(C++ function\), 577](#)
[esp_netif_set_ip4_addr \(C++ function\), 582](#)
[esp_netif_set_ip_info \(C++ function\), 578](#)
[esp_netif_set_mac \(C++ function\), 577](#)
[esp_netif_set_old_ip_info \(C++ function\), 579](#)
[esp_netif_transmit \(C++ function\), 587](#)
[esp_nimble_hci_and_controller_deinit \(C++ function\), 276](#)
[esp_nimble_hci_and_controller_init \(C++ function\), 276](#)

- `esp_nimble_hci_deinit` (C++ function), 276
`esp_nimble_hci_init` (C++ function), 276
`esp_now_add_peer` (C++ function), 527
`esp_now_deinit` (C++ function), 526
`esp_now_del_peer` (C++ function), 527
`ESP_NOW_ETH_ALEN` (C macro), 529
`esp_now_fetch_peer` (C++ function), 527
`esp_now_get_peer` (C++ function), 527
`esp_now_get_peer_num` (C++ function), 528
`esp_now_get_version` (C++ function), 526
`esp_now_init` (C++ function), 526
`esp_now_is_peer_exist` (C++ function), 528
`ESP_NOW_KEY_LEN` (C macro), 529
`ESP_NOW_MAX_DATA_LEN` (C macro), 529
`ESP_NOW_MAX_ENCRYPT_PEER_NUM` (C macro), 529
`ESP_NOW_MAX_TOTAL_PEER_NUM` (C macro), 529
`esp_now_mod_peer` (C++ function), 527
`esp_now_peer_info` (C++ class), 528
`esp_now_peer_info::channel` (C++ member), 528
`esp_now_peer_info::encrypt` (C++ member), 528
`esp_now_peer_info::ifidx` (C++ member), 528
`esp_now_peer_info::lmk` (C++ member), 528
`esp_now_peer_info::peer_addr` (C++ member), 528
`esp_now_peer_info::priv` (C++ member), 529
`esp_now_peer_info_t` (C++ type), 529
`esp_now_peer_num` (C++ class), 529
`esp_now_peer_num::encrypt_num` (C++ member), 529
`esp_now_peer_num::total_num` (C++ member), 529
`esp_now_peer_num_t` (C++ type), 529
`esp_now_recv_cb_t` (C++ type), 529
`esp_now_register_recv_cb` (C++ function), 526
`esp_now_register_send_cb` (C++ function), 526
`esp_now_send` (C++ function), 526
`esp_now_send_cb_t` (C++ type), 530
`ESP_NOW_SEND_FAIL` (C++ enumerator), 530
`esp_now_send_status_t` (C++ enum), 530
`ESP_NOW_SEND_SUCCESS` (C++ enumerator), 530
`esp_now_set_pmk` (C++ function), 528
`esp_now_unregister_recv_cb` (C++ function), 526
`esp_now_unregister_send_cb` (C++ function), 526
`ESP_OK` (C macro), 1018
`ESP_OK_EFUSE_CNT` (C macro), 1017
`esp_ota_begin` (C++ function), 1184
`esp_ota_check_rollback_is_possible` (C++ function), 1187
`esp_ota_end` (C++ function), 1184
`esp_ota_erase_last_boot_app_partition` (C++ function), 1187
`esp_ota_get_app_description` (C++ function), 1183
`esp_ota_get_app_elf_sha256` (C++ function), 1184
`esp_ota_get_boot_partition` (C++ function), 1185
`esp_ota_get_last_invalid_partition` (C++ function), 1186
`esp_ota_get_next_update_partition` (C++ function), 1185
`esp_ota_get_partition_description` (C++ function), 1186
`esp_ota_get_running_partition` (C++ function), 1185
`esp_ota_get_state_partition` (C++ function), 1186
`esp_ota_handle_t` (C++ type), 1187
`esp_ota_mark_app_invalid_rollback_and_reboot` (C++ function), 1186
`esp_ota_mark_app_valid_cancel_rollback` (C++ function), 1186
`esp_ota_set_boot_partition` (C++ function), 1185
`esp_ota_write` (C++ function), 1184
`esp_partition_check_identity` (C++ function), 977
`esp_partition_deregister_external` (C++ function), 977
`esp_partition_erase_range` (C++ function), 976
`esp_partition_find` (C++ function), 974
`esp_partition_find_first` (C++ function), 975
`esp_partition_get` (C++ function), 975
`esp_partition_get_sha256` (C++ function), 977
`esp_partition_iterator_release` (C++ function), 975
`esp_partition_iterator_t` (C++ type), 978
`esp_partition_mmap` (C++ function), 976
`esp_partition_next` (C++ function), 975
`esp_partition_read` (C++ function), 975
`esp_partition_register_external` (C++ function), 977
`ESP_PARTITION_SUBTYPE_ANY` (C++ enumerator), 980
`ESP_PARTITION_SUBTYPE_APP_FACTORY` (C++ enumerator), 979
`ESP_PARTITION_SUBTYPE_APP_OTA_0` (C++ enumerator), 979
`ESP_PARTITION_SUBTYPE_APP_OTA_1` (C++ enumerator), 979
`ESP_PARTITION_SUBTYPE_APP_OTA_10` (C++ enumerator), 979
`ESP_PARTITION_SUBTYPE_APP_OTA_11` (C++ enumerator), 979

- ESP_PARTITION_SUBTYPE_APP_OTA_12 (C++ enumerator), 979
- ESP_PARTITION_SUBTYPE_APP_OTA_13 (C++ enumerator), 979
- ESP_PARTITION_SUBTYPE_APP_OTA_14 (C++ enumerator), 979
- ESP_PARTITION_SUBTYPE_APP_OTA_15 (C++ enumerator), 979
- ESP_PARTITION_SUBTYPE_APP_OTA_2 (C++ enumerator), 979
- ESP_PARTITION_SUBTYPE_APP_OTA_3 (C++ enumerator), 979
- ESP_PARTITION_SUBTYPE_APP_OTA_4 (C++ enumerator), 979
- ESP_PARTITION_SUBTYPE_APP_OTA_5 (C++ enumerator), 979
- ESP_PARTITION_SUBTYPE_APP_OTA_6 (C++ enumerator), 979
- ESP_PARTITION_SUBTYPE_APP_OTA_7 (C++ enumerator), 979
- ESP_PARTITION_SUBTYPE_APP_OTA_8 (C++ enumerator), 979
- ESP_PARTITION_SUBTYPE_APP_OTA_9 (C++ enumerator), 979
- ESP_PARTITION_SUBTYPE_APP_OTA_MAX (C++ enumerator), 979
- ESP_PARTITION_SUBTYPE_APP_OTA_MIN (C++ enumerator), 979
- ESP_PARTITION_SUBTYPE_APP_TEST (C++ enumerator), 979
- ESP_PARTITION_SUBTYPE_DATA_COREDUMP (C++ enumerator), 980
- ESP_PARTITION_SUBTYPE_DATA_EFUSE_EM (C++ enumerator), 980
- ESP_PARTITION_SUBTYPE_DATA_ESPHTTPD (C++ enumerator), 980
- ESP_PARTITION_SUBTYPE_DATA_FAT (C++ enumerator), 980
- ESP_PARTITION_SUBTYPE_DATA_NVS (C++ enumerator), 980
- ESP_PARTITION_SUBTYPE_DATA_NVS_KEYS (C++ enumerator), 980
- ESP_PARTITION_SUBTYPE_DATA_OTA (C++ enumerator), 979
- ESP_PARTITION_SUBTYPE_DATA_PHY (C++ enumerator), 980
- ESP_PARTITION_SUBTYPE_DATA_SPIFFS (C++ enumerator), 980
- ESP_PARTITION_SUBTYPE_DATA_UNDEFINED (C++ enumerator), 980
- ESP_PARTITION_SUBTYPE_OTA (C macro), 978
- esp_partition_subtype_t (C++ enum), 979
- esp_partition_t (C++ class), 978
- esp_partition_t::address (C++ member), 978
- esp_partition_t::encrypted (C++ member), 978
- esp_partition_t::flash_chip (C++ member), 978
- esp_partition_t::label (C++ member), 978
- esp_partition_t::size (C++ member), 978
- esp_partition_t::subtype (C++ member), 978
- esp_partition_t::type (C++ member), 978
- ESP_PARTITION_TYPE_APP (C++ enumerator), 978
- ESP_PARTITION_TYPE_DATA (C++ enumerator), 978
- esp_partition_type_t (C++ enum), 978
- esp_partition_verify (C++ function), 975
- esp_partition_write (C++ function), 976
- ESP_PD_DOMAIN_MAX (C++ enumerator), 1203
- ESP_PD_DOMAIN_RTC_FAST_MEM (C++ enumerator), 1203
- ESP_PD_DOMAIN_RTC_PERIPH (C++ enumerator), 1202
- ESP_PD_DOMAIN_RTC_SLOW_MEM (C++ enumerator), 1202
- ESP_PD_DOMAIN_XTAL (C++ enumerator), 1203
- ESP_PD_OPTION_AUTO (C++ enumerator), 1203
- ESP_PD_OPTION_OFF (C++ enumerator), 1203
- ESP_PD_OPTION_ON (C++ enumerator), 1203
- esp_ping_callbacks_t (C++ class), 857
- esp_ping_callbacks_t::cb_args (C++ member), 857
- esp_ping_callbacks_t::on_ping_end (C++ member), 857
- esp_ping_callbacks_t::on_ping_success (C++ member), 857
- esp_ping_callbacks_t::on_ping_timeout (C++ member), 857
- esp_ping_config_t (C++ class), 857
- esp_ping_config_t::count (C++ member), 858
- esp_ping_config_t::data_size (C++ member), 858
- esp_ping_config_t::interval_ms (C++ member), 858
- esp_ping_config_t::target_addr (C++ member), 858
- esp_ping_config_t::task_prio (C++ member), 858
- esp_ping_config_t::task_stack_size (C++ member), 858
- esp_ping_config_t::timeout_ms (C++ member), 858
- esp_ping_config_t::tos (C++ member), 858
- ESP_PING_COUNT_INFINITE (C macro), 858
- ESP_PING_DEFAULT_CONFIG (C macro), 858
- esp_ping_delete_session (C++ function), 856
- esp_ping_get_profile (C++ function), 857
- esp_ping_handle_t (C++ type), 858
- esp_ping_new_session (C++ function), 856
- ESP_PING_PROF_DURATION (C++ enumerator), 858
- ESP_PING_PROF_IPADDR (C++ enumerator), 858

- ESP_PING_PROF_REPLY (C++ enumerator), 858
- ESP_PING_PROF_REQUEST (C++ enumerator), 858
- ESP_PING_PROF_SEQNO (C++ enumerator), 858
- ESP_PING_PROF_SIZE (C++ enumerator), 858
- ESP_PING_PROF_TIMEGAP (C++ enumerator), 858
- ESP_PING_PROF_TTL (C++ enumerator), 858
- esp_ping_profile_t (C++ enum), 858
- esp_ping_start (C++ function), 857
- esp_ping_stop (C++ function), 857
- ESP_PM_APB_FREQ_MAX (C++ enumerator), 1195
- esp_pm_config_esp32_t (C++ class), 1195
- esp_pm_config_esp32_t::light_sleep_enable (C++ member), 1195
- esp_pm_config_esp32_t::max_freq_mhz (C++ member), 1195
- esp_pm_config_esp32_t::min_freq_mhz (C++ member), 1195
- esp_pm_configure (C++ function), 1193
- ESP_PM_CPU_FREQ_MAX (C++ enumerator), 1195
- esp_pm_dump_locks (C++ function), 1194
- esp_pm_lock_acquire (C++ function), 1194
- esp_pm_lock_create (C++ function), 1193
- esp_pm_lock_delete (C++ function), 1194
- esp_pm_lock_handle_t (C++ type), 1195
- esp_pm_lock_release (C++ function), 1194
- esp_pm_lock_type_t (C++ enum), 1195
- ESP_PM_NO_LIGHT_SLEEP (C++ enumerator), 1195
- esp_power_level_t (C++ enum), 119
- esp_pthread_cfg_t (C++ class), 1023
- esp_pthread_cfg_t::inherit_cfg (C++ member), 1023
- esp_pthread_cfg_t::pin_to_core (C++ member), 1024
- esp_pthread_cfg_t::prio (C++ member), 1023
- esp_pthread_cfg_t::stack_size (C++ member), 1023
- esp_pthread_cfg_t::thread_name (C++ member), 1023
- esp_pthread_get_cfg (C++ function), 1023
- esp_pthread_get_default_config (C++ function), 1023
- esp_pthread_set_cfg (C++ function), 1023
- ESP_PWR_LVL_N0 (C++ enumerator), 119
- ESP_PWR_LVL_N11 (C++ enumerator), 120
- ESP_PWR_LVL_N12 (C++ enumerator), 119
- ESP_PWR_LVL_N14 (C++ enumerator), 120
- ESP_PWR_LVL_N2 (C++ enumerator), 120
- ESP_PWR_LVL_N3 (C++ enumerator), 119
- ESP_PWR_LVL_N5 (C++ enumerator), 120
- ESP_PWR_LVL_N6 (C++ enumerator), 119
- ESP_PWR_LVL_N8 (C++ enumerator), 120
- ESP_PWR_LVL_N9 (C++ enumerator), 119
- ESP_PWR_LVL_P1 (C++ enumerator), 120
- ESP_PWR_LVL_P3 (C++ enumerator), 119
- ESP_PWR_LVL_P4 (C++ enumerator), 120
- ESP_PWR_LVL_P6 (C++ enumerator), 120
- ESP_PWR_LVL_P7 (C++ enumerator), 120
- ESP_PWR_LVL_P9 (C++ enumerator), 120
- esp_random (C++ function), 1174
- esp_read_mac (C++ function), 1175
- esp_register_freertos_idle_hook (C++ function), 1130
- esp_register_freertos_idle_hook_for_cpu (C++ function), 1130
- esp_register_freertos_tick_hook (C++ function), 1131
- esp_register_freertos_tick_hook_for_cpu (C++ function), 1131
- esp_register_shutdown_handler (C++ function), 1174
- esp_reset_reason (C++ function), 1174
- esp_reset_reason_t (C++ enum), 1177
- esp_restart (C++ function), 1174
- ESP_RST_BROWNOUT (C++ enumerator), 1177
- ESP_RST_DEEPSLEEP (C++ enumerator), 1177
- ESP_RST_EXT (C++ enumerator), 1177
- ESP_RST_INT_WDT (C++ enumerator), 1177
- ESP_RST_PANIC (C++ enumerator), 1177
- ESP_RST_POWERON (C++ enumerator), 1177
- ESP_RST_SDIO (C++ enumerator), 1177
- ESP_RST_SW (C++ enumerator), 1177
- ESP_RST_TASK_WDT (C++ enumerator), 1177
- ESP_RST_UNKNOWN (C++ enumerator), 1177
- ESP_RST_WDT (C++ enumerator), 1177
- ESP_SCO_DATA_PATH_HCI (C++ enumerator), 120
- ESP_SCO_DATA_PATH_PCM (C++ enumerator), 120
- esp_sco_data_path_t (C++ enum), 120
- esp_service_source_t (C++ enum), 162
- esp_set_deep_sleep_wake_stub (C++ function), 1202
- esp_sleep_disable_wakeup_source (C++ function), 1199
- esp_sleep_enable_ext0_wakeup (C++ function), 1200
- esp_sleep_enable_ext1_wakeup (C++ function), 1200
- esp_sleep_enable_gpio_wakeup (C++ function), 1200
- esp_sleep_enable_timer_wakeup (C++ function), 1199
- esp_sleep_enable_touchpad_wakeup (C++ function), 1199
- esp_sleep_enable_uart_wakeup (C++ function), 1201
- esp_sleep_enable_ulp_wakeup (C++ function), 1199
- esp_sleep_ext1_wakeup_mode_t (C++ enum), 1202
- esp_sleep_get_ext1_wakeup_status (C++ function), 1201
- esp_sleep_get_touchpad_wakeup_status (C++ function), 1199
- esp_sleep_get_wakeup_cause (C++ function), 1202

- esp_sleep_pd_config (C++ function), 1201
 esp_sleep_pd_domain_t (C++ enum), 1202
 esp_sleep_pd_option_t (C++ enum), 1203
 esp_sleep_source_t (C++ enum), 1203
 ESP_SLEEP_WAKEUP_ALL (C++ enumerator), 1203
 esp_sleep_wakeup_cause_t (C++ type), 1202
 ESP_SLEEP_WAKEUP_EXT0 (C++ enumerator), 1203
 ESP_SLEEP_WAKEUP_EXT1 (C++ enumerator), 1203
 ESP_SLEEP_WAKEUP_GPIO (C++ enumerator), 1203
 ESP_SLEEP_WAKEUP_TIMER (C++ enumerator), 1203
 ESP_SLEEP_WAKEUP_TOUCHPAD (C++ enumerator), 1203
 ESP_SLEEP_WAKEUP_UART (C++ enumerator), 1203
 ESP_SLEEP_WAKEUP_ULP (C++ enumerator), 1203
 ESP_SLEEP_WAKEUP_UNDEFINED (C++ enumerator), 1203
 esp_smartconfig_fast_mode (C++ function), 523
 esp_smartconfig_get_version (C++ function), 522
 esp_smartconfig_set_type (C++ function), 523
 esp_smartconfig_start (C++ function), 522
 esp_smartconfig_stop (C++ function), 522
 esp_spiffs_format (C++ function), 984
 esp_spiffs_info (C++ function), 984
 esp_spiffs_mounted (C++ function), 984
 ESP_SPP_BUSY (C++ enumerator), 245
 esp_spp_cb_event_t (C++ enum), 245
 esp_spp_cb_param_t (C++ union), 240
 esp_spp_cb_param_t::cl_init (C++ member), 241
 esp_spp_cb_param_t::close (C++ member), 240
 esp_spp_cb_param_t::cong (C++ member), 241
 esp_spp_cb_param_t::data_ind (C++ member), 241
 esp_spp_cb_param_t::disc_comp (C++ member), 240
 esp_spp_cb_param_t::init (C++ member), 240
 esp_spp_cb_param_t::open (C++ member), 240
 esp_spp_cb_param_t::spp_cl_init_evt_param (C++ class), 241
 esp_spp_cb_param_t::spp_cl_init_evt_param::handle (C++ member), 241
 esp_spp_cb_param_t::spp_cl_init_evt_param::status (C++ member), 241
 esp_spp_cb_param_t::spp_cl_init_evt_param::status (C++ member), 241
 esp_spp_cb_param_t::spp_cl_init_evt_param::spp_cb (C++ member), 241
 esp_spp_cb_param_t::spp_close_evt_param (C++ class), 241
 esp_spp_cb_param_t::spp_close_evt_param::async (C++ member), 241
 esp_spp_cb_param_t::spp_close_evt_param::handle (C++ member), 241
 esp_spp_cb_param_t::spp_close_evt_param::port_status (C++ member), 241
 esp_spp_cb_param_t::spp_close_evt_param::status (C++ member), 241
 esp_spp_cb_param_t::spp_cong_evt_param (C++ class), 241
 esp_spp_cb_param_t::spp_cong_evt_param::cong (C++ member), 242
 esp_spp_cb_param_t::spp_cong_evt_param::handle (C++ member), 241
 esp_spp_cb_param_t::spp_cong_evt_param::status (C++ member), 241
 esp_spp_cb_param_t::spp_data_ind_evt_param (C++ class), 242
 esp_spp_cb_param_t::spp_data_ind_evt_param::data (C++ member), 242
 esp_spp_cb_param_t::spp_data_ind_evt_param::handle (C++ member), 242
 esp_spp_cb_param_t::spp_data_ind_evt_param::len (C++ member), 242
 esp_spp_cb_param_t::spp_data_ind_evt_param::status (C++ member), 242
 esp_spp_cb_param_t::spp_discovery_comp_evt_param (C++ class), 242
 esp_spp_cb_param_t::spp_discovery_comp_evt_param (C++ member), 242
 esp_spp_cb_param_t::spp_discovery_comp_evt_param (C++ member), 242
 esp_spp_cb_param_t::spp_discovery_comp_evt_param (C++ member), 242
 esp_spp_cb_param_t::spp_discovery_comp_evt_param (C++ member), 242
 esp_spp_cb_param_t::spp_discovery_comp_evt_param (C++ member), 242
 esp_spp_cb_param_t::spp_discovery_comp_evt_param (C++ member), 242
 esp_spp_cb_param_t::spp_init_evt_param (C++ class), 242
 esp_spp_cb_param_t::spp_init_evt_param::status (C++ member), 242
 esp_spp_cb_param_t::spp_open_evt_param (C++ class), 242
 esp_spp_cb_param_t::spp_open_evt_param::fd (C++ member), 242
 esp_spp_cb_param_t::spp_open_evt_param::handle (C++ member), 242
 esp_spp_cb_param_t::spp_open_evt_param::rem_bda (C++ member), 242
 esp_spp_cb_param_t::spp_open_evt_param::status (C++ member), 242
 esp_spp_cb_param_t::spp_srv_open_evt_param (C++ class), 243
 esp_spp_cb_param_t::spp_srv_open_evt_param::fd (C++ member), 243
 esp_spp_cb_param_t::spp_srv_open_evt_param::handle (C++ member), 243

- `esp_timer_cb_t` (C++ type), 1156
- `esp_timer_create` (C++ function), 1154
- `esp_timer_create_args_t` (C++ class), 1155
- `esp_timer_create_args_t::arg` (C++ member), 1156
- `esp_timer_create_args_t::callback` (C++ member), 1156
- `esp_timer_create_args_t::dispatch_method` (C++ member), 1156
- `esp_timer_create_args_t::name` (C++ member), 1156
- `esp_timer_deinit` (C++ function), 1154
- `esp_timer_delete` (C++ function), 1155
- `esp_timer_dispatch_t` (C++ enum), 1156
- `esp_timer_dump` (C++ function), 1155
- `esp_timer_get_next_alarm` (C++ function), 1155
- `esp_timer_get_time` (C++ function), 1155
- `esp_timer_handle_t` (C++ type), 1156
- `esp_timer_init` (C++ function), 1154
- `esp_timer_start_once` (C++ function), 1154
- `esp_timer_start_periodic` (C++ function), 1154
- `esp_timer_stop` (C++ function), 1154
- `ESP_TIMER_TASK` (C++ enumerator), 1156
- `esp_tls` (C++ class), 817
- `esp_tls::conn_state` (C++ member), 818
- `esp_tls::error_handle` (C++ member), 818
- `esp_tls::is_tls` (C++ member), 818
- `esp_tls::read` (C++ member), 818
- `esp_tls::role` (C++ member), 818
- `esp_tls::rset` (C++ member), 818
- `esp_tls::sockfd` (C++ member), 818
- `esp_tls::write` (C++ member), 818
- `esp_tls::wset` (C++ member), 818
- `esp_tls_cfg` (C++ class), 816
- `esp_tls_cfg::alpn_protos` (C++ member), 816
- `esp_tls_cfg::cacert_buf` (C++ member), 816
- `esp_tls_cfg::cacert_bytes` (C++ member), 817
- `esp_tls_cfg::cacert_pem_buf` (C++ member), 816
- `esp_tls_cfg::cacert_pem_bytes` (C++ member), 817
- `esp_tls_cfg::clientcert_buf` (C++ member), 817
- `esp_tls_cfg::clientcert_bytes` (C++ member), 817
- `esp_tls_cfg::clientcert_pem_buf` (C++ member), 817
- `esp_tls_cfg::clientcert_pem_bytes` (C++ member), 817
- `esp_tls_cfg::clientkey_buf` (C++ member), 817
- `esp_tls_cfg::clientkey_bytes` (C++ member), 817
- `esp_tls_cfg::clientkey_password` (C++ member), 817
- `esp_tls_cfg::clientkey_password_len` (C++ member), 817
- `esp_tls_cfg::clientkey_pem_buf` (C++ member), 817
- `esp_tls_cfg::clientkey_pem_bytes` (C++ member), 817
- `esp_tls_cfg::common_name` (C++ member), 817
- `esp_tls_cfg::keep_alive_cfg` (C++ member), 817
- `esp_tls_cfg::non_block` (C++ member), 817
- `esp_tls_cfg::psk_hint_key` (C++ member), 817
- `esp_tls_cfg::skip_common_name` (C++ member), 817
- `esp_tls_cfg::timeout_ms` (C++ member), 817
- `esp_tls_cfg::use_global_ca_store` (C++ member), 817
- `esp_tls_cfg_t` (C++ type), 819
- `ESP_TLS_CLIENT` (C++ enumerator), 820
- `esp_tls_conn_delete` (C++ function), 814
- `esp_tls_conn_http_new` (C++ function), 813
- `esp_tls_conn_http_new_async` (C++ function), 814
- `esp_tls_conn_new` (C++ function), 812
- `esp_tls_conn_new_async` (C++ function), 813
- `esp_tls_conn_new_sync` (C++ function), 813
- `esp_tls_conn_read` (C++ function), 814
- `esp_tls_conn_state` (C++ enum), 819
- `esp_tls_conn_state_t` (C++ type), 819
- `esp_tls_conn_write` (C++ function), 814
- `ESP_TLS_CONNECTING` (C++ enumerator), 820
- `ESP_TLS_DONE` (C++ enumerator), 820
- `esp_tls_error_handle_t` (C++ type), 819
- `ESP_TLS_FAIL` (C++ enumerator), 820
- `esp_tls_free_global_ca_store` (C++ function), 815
- `esp_tls_get_and_clear_last_error` (C++ function), 815
- `esp_tls_get_bytes_avail` (C++ function), 814
- `ESP_TLS_HANDSHAKE` (C++ enumerator), 820
- `ESP_TLS_INIT` (C++ enumerator), 819
- `esp_tls_init` (C++ function), 812
- `esp_tls_init_global_ca_store` (C++ function), 815
- `esp_tls_last_error` (C++ class), 815
- `esp_tls_last_error::esp_tls_error_code` (C++ member), 815
- `esp_tls_last_error::esp_tls_flags` (C++ member), 816
- `esp_tls_last_error::last_error` (C++ member), 815
- `esp_tls_last_error_t` (C++ type), 819
- `esp_tls_role` (C++ enum), 820
- `esp_tls_role_t` (C++ type), 819
- `ESP_TLS_SERVER` (C++ enumerator), 820

- esp_tls_set_global_ca_store (C++ function), 815
 esp_tls_t (C++ type), 819
 esp_unregister_shutdown_handler (C++ function), 1174
 ESP_UUID_LEN_128 (C macro), 121
 ESP_UUID_LEN_16 (C macro), 121
 ESP_UUID_LEN_32 (C macro), 121
 esp_vendor_ie_cb_t (C++ type), 504
 esp_vfs_close (C++ function), 989
 esp_vfs_dev_uart_port_set_rx_line_endings (C++ function), 993
 esp_vfs_dev_uart_port_set_tx_line_endings (C++ function), 993
 esp_vfs_dev_uart_register (C++ function), 993
 esp_vfs_dev_uart_set_rx_line_endings (C++ function), 993
 esp_vfs_dev_uart_set_tx_line_endings (C++ function), 993
 esp_vfs_dev_uart_use_driver (C++ function), 994
 esp_vfs_dev_uart_use_nonblocking (C++ function), 993
 esp_vfs_fat_mount_config_t (C++ class), 921, 995
 esp_vfs_fat_mount_config_t::allocation_size (C++ member), 921, 996
 esp_vfs_fat_mount_config_t::format_if_mount_failed (C++ member), 921, 996
 esp_vfs_fat_mount_config_t::max_files (C++ member), 921, 996
 esp_vfs_fat_rawflash_mount (C++ function), 922
 esp_vfs_fat_rawflash_unmount (C++ function), 922
 esp_vfs_fat_register (C++ function), 920
 esp_vfs_fat_sdmmc_mount (C++ function), 920
 esp_vfs_fat_sdmmc_unmount (C++ function), 921
 esp_vfs_fat_spiflash_mount (C++ function), 995
 esp_vfs_fat_spiflash_unmount (C++ function), 996
 esp_vfs_fat_unregister_path (C++ function), 920
 ESP_VFS_FLAG_CONTEXT_PTR (C macro), 992
 ESP_VFS_FLAG_DEFAULT (C macro), 992
 esp_vfs_fstat (C++ function), 989
 esp_vfs_id_t (C++ type), 992
 esp_vfs_link (C++ function), 989
 esp_vfs_lseek (C++ function), 989
 esp_vfs_open (C++ function), 989
 ESP_VFS_PATH_MAX (C macro), 992
 esp_vfs_poll (C++ function), 991
 esp_vfs_pread (C++ function), 991
 esp_vfs_pwrite (C++ function), 991
 esp_vfs_read (C++ function), 989
 esp_vfs_register (C++ function), 989
 esp_vfs_register_fd (C++ function), 990
 esp_vfs_register_fd_range (C++ function), 989
 esp_vfs_register_with_id (C++ function), 989
 esp_vfs_rename (C++ function), 989
 esp_vfs_select (C++ function), 990
 esp_vfs_select_sem_t (C++ class), 991
 esp_vfs_select_sem_t::is_sem_local (C++ member), 991
 esp_vfs_select_sem_t::sem (C++ member), 991
 esp_vfs_select_triggered (C++ function), 990
 esp_vfs_select_triggered_isr (C++ function), 991
 esp_vfs_spiffs_conf_t (C++ class), 984
 esp_vfs_spiffs_conf_t::base_path (C++ member), 984
 esp_vfs_spiffs_conf_t::format_if_mount_failed (C++ member), 985
 esp_vfs_spiffs_conf_t::max_files (C++ member), 985
 esp_vfs_spiffs_conf_t::partition_label (C++ member), 984
 esp_vfs_spiffs_register (C++ function), 984
 esp_vfs_spiffs_unregister (C++ function), 984
 esp_vfs_stat (C++ function), 989
 esp_vfs_t (C++ class), 991
 esp_vfs_t::end_select (C++ member), 992
 esp_vfs_t::flags (C++ member), 992
 esp_vfs_t::get_socket_select_semaphore (C++ member), 992
 esp_vfs_t::socket_select (C++ member), 992
 esp_vfs_t::start_select (C++ member), 992
 esp_vfs_t::stop_socket_select (C++ member), 992
 esp_vfs_t::stop_socket_select_isr (C++ member), 992
 esp_vfs_unlink (C++ function), 989
 esp_vfs_unregister (C++ function), 990
 esp_vfs_unregister_fd (C++ function), 990
 esp_vfs_utime (C++ function), 989
 esp_vfs_write (C++ function), 989
 esp_vhci_host_callback (C++ class), 117
 esp_vhci_host_callback::notify_host_recv (C++ member), 117
 esp_vhci_host_callback::notify_host_send_available (C++ member), 117
 esp_vhci_host_callback_t (C++ type), 118
 esp_vhci_host_check_send_available (C++ function), 114
 esp_vhci_host_register_callback (C++ function), 115
 esp_vhci_host_send_packet (C++ function),

- 115
- `esp_wake_deep_sleep` (C++ function), 1202
- `esp_websocket_client_config_t` (C++ class), 882
- `esp_websocket_client_config_t::buffer_size` (C++ member), 883
- `esp_websocket_client_config_t::cert_pem` (C++ member), 883
- `esp_websocket_client_config_t::disable_ssl` (C++ member), 883
- `esp_websocket_client_config_t::headers` (C++ member), 883
- `esp_websocket_client_config_t::host` (C++ member), 882
- `esp_websocket_client_config_t::keep_alive` (C++ member), 883
- `esp_websocket_client_config_t::keep_alive_enable` (C++ member), 883
- `esp_websocket_client_config_t::keep_alive_idle` (C++ member), 883
- `esp_websocket_client_config_t::keep_alive_interval` (C++ member), 883
- `esp_websocket_client_config_t::password` (C++ member), 883
- `esp_websocket_client_config_t::path` (C++ member), 883
- `esp_websocket_client_config_t::port` (C++ member), 882
- `esp_websocket_client_config_t::subprotocol` (C++ member), 883
- `esp_websocket_client_config_t::task_priority` (C++ member), 883
- `esp_websocket_client_config_t::task_stack` (C++ member), 883
- `esp_websocket_client_config_t::transport` (C++ member), 883
- `esp_websocket_client_config_t::uri` (C++ member), 882
- `esp_websocket_client_config_t::user_agent` (C++ member), 883
- `esp_websocket_client_config_t::user_context` (C++ member), 883
- `esp_websocket_client_config_t::username` (C++ member), 882
- `esp_websocket_client_destroy` (C++ function), 881
- `esp_websocket_client_handle_t` (C++ type), 883
- `esp_websocket_client_init` (C++ function), 880
- `esp_websocket_client_is_connected` (C++ function), 881
- `esp_websocket_client_send` (C++ function), 881
- `esp_websocket_client_send_bin` (C++ function), 881
- `esp_websocket_client_send_text` (C++ function), 881
- `esp_websocket_client_set_uri` (C++ function), 880
- `esp_websocket_client_start` (C++ function), 880
- `esp_websocket_client_stop` (C++ function), 881
- `esp_websocket_event_data_t` (C++ class), 882
- `esp_websocket_event_data_t::client` (C++ member), 882
- `esp_websocket_event_data_t::data_len` (C++ member), 882
- `esp_websocket_event_data_t::data_ptr` (C++ member), 882
- `esp_websocket_event_data_t::op_code` (C++ member), 882
- `esp_websocket_event_data_t::payload_len` (C++ member), 882
- `esp_websocket_event_data_t::payload_offset` (C++ member), 882
- `esp_websocket_event_data_t::user_context` (C++ member), 882
- `esp_websocket_event_id_t` (C++ enum), 883
- `esp_websocket_register_events` (C++ function), 882
- `esp_websocket_transport_t` (C++ enum), 884
- `esp_wifi_80211_tx` (C++ function), 499
- `esp_wifi_ap_get_sta_aid` (C++ function), 497
- `esp_wifi_ap_get_sta_list` (C++ function), 497
- `esp_wifi_clear_default_wifi_driver_and_handlers` (C++ function), 584
- `esp_wifi_clear_fast_connect` (C++ function), 491
- `esp_wifi_connect` (C++ function), 491
- `esp_wifi_death_sta` (C++ function), 491
- `esp_wifi_deinit` (C++ function), 490
- `esp_wifi_disconnect` (C++ function), 491
- `esp_wifi_get_ant` (C++ function), 500
- `esp_wifi_get_ant_gpio` (C++ function), 500
- `esp_wifi_get_bandwidth` (C++ function), 494
- `esp_wifi_get_channel` (C++ function), 494
- `esp_wifi_get_config` (C++ function), 497
- `esp_wifi_get_country` (C++ function), 495
- `esp_wifi_get_event_mask` (C++ function), 499
- `esp_wifi_get_inactive_time` (C++ function), 501
- `esp_wifi_get_mac` (C++ function), 495
- `esp_wifi_get_max_tx_power` (C++ function), 498
- `esp_wifi_get_mode` (C++ function), 490
- `esp_wifi_get_promiscuous` (C++ function), 496
- `esp_wifi_get_promiscuous_ctrl_filter` (C++ function), 496
- `esp_wifi_get_promiscuous_filter` (C++ function), 496
- `esp_wifi_get_protocol` (C++ function), 493

- esp_wifi_get_ps (C++ function), 493
- esp_wifi_init (C++ function), 490
- ESP_WIFI_MAX_CONN_NUM (C macro), 514
- esp_wifi_restore (C++ function), 491
- esp_wifi_scan_get_ap_num (C++ function), 492
- esp_wifi_scan_get_ap_records (C++ function), 492
- esp_wifi_scan_start (C++ function), 492
- esp_wifi_scan_stop (C++ function), 492
- esp_wifi_set_ant (C++ function), 500
- esp_wifi_set_ant_gpio (C++ function), 500
- esp_wifi_set_bandwidth (C++ function), 493
- esp_wifi_set_channel (C++ function), 494
- esp_wifi_set_config (C++ function), 496
- esp_wifi_set_country (C++ function), 494
- esp_wifi_set_csi (C++ function), 500
- esp_wifi_set_csi_config (C++ function), 499
- esp_wifi_set_csi_rx_cb (C++ function), 499
- esp_wifi_set_default_wifi_ap_handlers (C++ function), 584
- esp_wifi_set_default_wifi_sta_handlers (C++ function), 584
- esp_wifi_set_event_mask (C++ function), 499
- esp_wifi_set_inactive_time (C++ function), 500
- esp_wifi_set_mac (C++ function), 495
- esp_wifi_set_max_tx_power (C++ function), 498
- esp_wifi_set_mode (C++ function), 490
- esp_wifi_set_promiscuous (C++ function), 495
- esp_wifi_set_promiscuous_ctrl_filter (C++ function), 496
- esp_wifi_set_promiscuous_filter (C++ function), 496
- esp_wifi_set_promiscuous_rx_cb (C++ function), 495
- esp_wifi_set_protocol (C++ function), 493
- esp_wifi_set_ps (C++ function), 493
- esp_wifi_set_storage (C++ function), 497
- esp_wifi_set_vendor_ie (C++ function), 498
- esp_wifi_set_vendor_ie_cb (C++ function), 498
- esp_wifi_sta_get_ap_info (C++ function), 492
- esp_wifi_start (C++ function), 490
- esp_wifi_status_dump (C++ function), 501
- esp_wifi_stop (C++ function), 490
- essl_clear_intr (C++ function), 888
- essl_get_intr (C++ function), 888
- essl_get_intr_ena (C++ function), 888
- essl_get_packet (C++ function), 887
- essl_get_rx_data_size (C++ function), 886
- essl_get_tx_buffer_num (C++ function), 886
- essl_handle_t (C++ type), 889
- essl_init (C++ function), 886
- essl_read_reg (C++ function), 887
- essl_reset_cnt (C++ function), 886
- essl_sdio_config_t (C++ class), 889
- essl_sdio_config_t::card (C++ member), 889
- essl_sdio_config_t::recv_buffer_size (C++ member), 889
- essl_sdio_deinit_dev (C++ function), 889
- essl_sdio_init_dev (C++ function), 889
- essl_send_packet (C++ function), 886
- essl_send_slave_intr (C++ function), 888
- essl_set_intr_ena (C++ function), 888
- essl_wait_for_ready (C++ function), 886
- essl_wait_int (C++ function), 887
- essl_write_reg (C++ function), 887
- eStandardSleep (C++ enumerator), 1059
- eSuspended (C++ enumerator), 1058
- eTaskGetState (C++ function), 1043
- eTaskState (C++ enum), 1058
- ETH_CMD_G_MAC_ADDR (C++ enumerator), 564
- ETH_CMD_G_PHY_ADDR (C++ enumerator), 565
- ETH_CMD_G_SPEED (C++ enumerator), 565
- ETH_CMD_S_MAC_ADDR (C++ enumerator), 565
- ETH_CMD_S_PHY_ADDR (C++ enumerator), 565
- ETH_CMD_S_PROMISCUOUS (C++ enumerator), 565
- ETH_CRC_LEN (C macro), 564
- ETH_DEFAULT_CONFIG (C macro), 562
- ETH_DUPLEX_FULL (C++ enumerator), 565
- ETH_DUPLEX_HALF (C++ enumerator), 565
- eth_duplex_t (C++ enum), 565
- eth_event_t (C++ enum), 565
- ETH_HEADER_LEN (C macro), 564
- ETH_JUMBO_FRAME_PAYLOAD_LEN (C macro), 564
- ETH_LINK_DOWN (C++ enumerator), 565
- eth_link_t (C++ enum), 565
- ETH_LINK_UP (C++ enumerator), 565
- eth_mac_config_t (C++ class), 569
- eth_mac_config_t::flags (C++ member), 569
- eth_mac_config_t::rx_task_prio (C++ member), 569
- eth_mac_config_t::rx_task_stack_size (C++ member), 569
- eth_mac_config_t::smi_mdc_gpio_num (C++ member), 569
- eth_mac_config_t::smi_mdio_gpio_num (C++ member), 569
- eth_mac_config_t::sw_reset_timeout_ms (C++ member), 569
- ETH_MAC_DEFAULT_CONFIG (C macro), 569
- ETH_MAC_FLAG_PIN_TO_CORE (C macro), 569
- ETH_MAC_FLAG_WORK_WITH_CACHE_DISABLE (C macro), 569
- ETH_MAX_PACKET_SIZE (C macro), 564
- ETH_MAX_PAYLOAD_LEN (C macro), 564
- ETH_MIN_PACKET_SIZE (C macro), 564
- ETH_MIN_PAYLOAD_LEN (C macro), 564
- eth_phy_config_t (C++ class), 572

- eth_phy_config_t::autonego_timeout_ms (C++ member), 572
- eth_phy_config_t::phy_addr (C++ member), 572
- eth_phy_config_t::reset_gpio_num (C++ member), 572
- eth_phy_config_t::reset_timeout_ms (C++ member), 572
- ETH_PHY_DEFAULT_CONFIG (C macro), 572
- ETH_SPEED_100M (C++ enumerator), 565
- ETH_SPEED_10M (C++ enumerator), 565
- eth_speed_t (C++ enum), 565
- ETH_STATE_DEINIT (C++ enumerator), 564
- ETH_STATE_DUPLEX (C++ enumerator), 564
- ETH_STATE_LINK (C++ enumerator), 564
- ETH_STATE_LLIMIT (C++ enumerator), 564
- ETH_STATE_SPEED (C++ enumerator), 564
- ETH_VLAN_TAG_LEN (C macro), 564
- ETHERNET_EVENT_CONNECTED (C++ enumerator), 565
- ETHERNET_EVENT_DISCONNECTED (C++ enumerator), 565
- ETHERNET_EVENT_START (C++ enumerator), 565
- ETHERNET_EVENT_STOP (C++ enumerator), 565
- ETS_INTERNAL_INTR_SOURCE_OFF (C macro), 1166
- ETS_INTERNAL_PROFILING_INTR_SOURCE (C macro), 1166
- ETS_INTERNAL_SW0_INTR_SOURCE (C macro), 1166
- ETS_INTERNAL_SW1_INTR_SOURCE (C macro), 1166
- ETS_INTERNAL_TIMER0_INTR_SOURCE (C macro), 1166
- ETS_INTERNAL_TIMER1_INTR_SOURCE (C macro), 1166
- ETS_INTERNAL_TIMER2_INTR_SOURCE (C macro), 1166
- EventBits_t (C++ type), 1115
- EventGroupHandle_t (C++ type), 1115
- ## F
- FAST_PROV_ACT_ENTER (C++ enumerator), 325
- FAST_PROV_ACT_EXIT (C++ enumerator), 325
- FAST_PROV_ACT_MAX (C++ enumerator), 325
- FAST_PROV_ACT_NONE (C++ enumerator), 325
- FAST_PROV_ACT_SUSPEND (C++ enumerator), 325
- ff_diskio_impl_t (C++ class), 922
- ff_diskio_impl_t::init (C++ member), 923
- ff_diskio_impl_t::ioctl (C++ member), 923
- ff_diskio_impl_t::read (C++ member), 923
- ff_diskio_impl_t::status (C++ member), 923
- ff_diskio_impl_t::write (C++ member), 923
- ff_diskio_register (C++ function), 922
- ff_diskio_register_raw_partition (C++ function), 923
- ff_diskio_register_sdmmc (C++ function), 923
- ff_diskio_register_wl_partition (C++ function), 923
- ## G
- gpio_config (C++ function), 621
- gpio_config_t (C++ class), 626
- gpio_config_t::intr_type (C++ member), 626
- gpio_config_t::mode (C++ member), 626
- gpio_config_t::pin_bit_mask (C++ member), 626
- gpio_config_t::pull_down_en (C++ member), 626
- gpio_config_t::pull_up_en (C++ member), 626
- gpio_deep_sleep_hold_dis (C++ function), 625
- gpio_deep_sleep_hold_en (C++ function), 625
- GPIO_DRIVE_CAP_0 (C++ enumerator), 629
- GPIO_DRIVE_CAP_1 (C++ enumerator), 629
- GPIO_DRIVE_CAP_2 (C++ enumerator), 629
- GPIO_DRIVE_CAP_3 (C++ enumerator), 629
- GPIO_DRIVE_CAP_DEFAULT (C++ enumerator), 629
- GPIO_DRIVE_CAP_MAX (C++ enumerator), 629
- gpio_drive_cap_t (C++ enum), 629
- GPIO_FLOATING (C++ enumerator), 629
- gpio_get_drive_capability (C++ function), 624
- gpio_get_level (C++ function), 622
- gpio_hold_dis (C++ function), 625
- gpio_hold_en (C++ function), 625
- gpio_install_isr_service (C++ function), 623
- gpio_int_type_t (C++ enum), 628
- GPIO_INTR_ANYEDGE (C++ enumerator), 628
- GPIO_INTR_DISABLE (C++ enumerator), 628
- gpio_intr_disable (C++ function), 621
- gpio_intr_enable (C++ function), 621
- GPIO_INTR_HIGH_LEVEL (C++ enumerator), 628
- GPIO_INTR_LOW_LEVEL (C++ enumerator), 628
- GPIO_INTR_MAX (C++ enumerator), 628
- GPIO_INTR_NEGEDGE (C++ enumerator), 628
- GPIO_INTR_POSEDGE (C++ enumerator), 628
- gpio_iomux_in (C++ function), 625
- gpio_iomux_out (C++ function), 625
- gpio_isr_handle_t (C++ type), 626
- gpio_isr_handler_add (C++ function), 624
- gpio_isr_handler_remove (C++ function), 624
- gpio_isr_register (C++ function), 623
- gpio_isr_t (C++ type), 626
- GPIO_MODE_DISABLE (C++ enumerator), 628
- GPIO_MODE_INPUT (C++ enumerator), 628
- GPIO_MODE_INPUT_OUTPUT (C++ enumerator), 629

- GPIO_MODE_INPUT_OUTPUT_OD (C++ *enumerator*), 628
- GPIO_MODE_OUTPUT (C++ *enumerator*), 628
- GPIO_MODE_OUTPUT_OD (C++ *enumerator*), 628
- gpio_mode_t (C++ *enum*), 628
- GPIO_NUM_0 (C++ *enumerator*), 626
- GPIO_NUM_1 (C++ *enumerator*), 626
- GPIO_NUM_10 (C++ *enumerator*), 627
- GPIO_NUM_11 (C++ *enumerator*), 627
- GPIO_NUM_12 (C++ *enumerator*), 627
- GPIO_NUM_13 (C++ *enumerator*), 627
- GPIO_NUM_14 (C++ *enumerator*), 627
- GPIO_NUM_15 (C++ *enumerator*), 627
- GPIO_NUM_16 (C++ *enumerator*), 627
- GPIO_NUM_17 (C++ *enumerator*), 627
- GPIO_NUM_18 (C++ *enumerator*), 627
- GPIO_NUM_19 (C++ *enumerator*), 627
- GPIO_NUM_2 (C++ *enumerator*), 626
- GPIO_NUM_20 (C++ *enumerator*), 627
- GPIO_NUM_21 (C++ *enumerator*), 627
- GPIO_NUM_26 (C++ *enumerator*), 627
- GPIO_NUM_27 (C++ *enumerator*), 627
- GPIO_NUM_28 (C++ *enumerator*), 627
- GPIO_NUM_29 (C++ *enumerator*), 627
- GPIO_NUM_3 (C++ *enumerator*), 626
- GPIO_NUM_30 (C++ *enumerator*), 627
- GPIO_NUM_31 (C++ *enumerator*), 628
- GPIO_NUM_32 (C++ *enumerator*), 628
- GPIO_NUM_33 (C++ *enumerator*), 628
- GPIO_NUM_34 (C++ *enumerator*), 628
- GPIO_NUM_35 (C++ *enumerator*), 628
- GPIO_NUM_36 (C++ *enumerator*), 628
- GPIO_NUM_37 (C++ *enumerator*), 628
- GPIO_NUM_38 (C++ *enumerator*), 628
- GPIO_NUM_39 (C++ *enumerator*), 628
- GPIO_NUM_4 (C++ *enumerator*), 627
- GPIO_NUM_5 (C++ *enumerator*), 627
- GPIO_NUM_6 (C++ *enumerator*), 627
- GPIO_NUM_7 (C++ *enumerator*), 627
- GPIO_NUM_8 (C++ *enumerator*), 627
- GPIO_NUM_9 (C++ *enumerator*), 627
- GPIO_NUM_MAX (C++ *enumerator*), 628
- GPIO_NUM_NC (C++ *enumerator*), 626
- gpio_num_t (C++ *enum*), 626
- GPIO_PORT_0 (C++ *enumerator*), 626
- GPIO_PORT_MAX (C++ *enumerator*), 626
- gpio_port_t (C++ *enum*), 626
- gpio_pull_mode_t (C++ *enum*), 629
- gpio_pulldown_dis (C++ *function*), 623
- GPIO_PULLDOWN_DISABLE (C++ *enumerator*), 629
- gpio_pulldown_en (C++ *function*), 623
- GPIO_PULLDOWN_ENABLE (C++ *enumerator*), 629
- GPIO_PULLDOWN_ONLY (C++ *enumerator*), 629
- gpio_pulldown_t (C++ *enum*), 629
- gpio_pullup_dis (C++ *function*), 623
- GPIO_PULLUP_DISABLE (C++ *enumerator*), 629
- gpio_pullup_en (C++ *function*), 623
- GPIO_PULLUP_ENABLE (C++ *enumerator*), 629
- GPIO_PULLUP_ONLY (C++ *enumerator*), 629
- GPIO_PULLUP_PULLDOWN (C++ *enumerator*), 629
- gpio_pullup_t (C++ *enum*), 629
- gpio_reset_pin (C++ *function*), 621
- gpio_set_direction (C++ *function*), 622
- gpio_set_drive_capability (C++ *function*), 624
- gpio_set_intr_type (C++ *function*), 621
- gpio_set_level (C++ *function*), 622
- gpio_set_pull_mode (C++ *function*), 622
- gpio_uninstall_isr_service (C++ *function*), 624
- gpio_wakeup_disable (C++ *function*), 622
- gpio_wakeup_enable (C++ *function*), 622
- ## H
- hall_sensor_read (C++ *function*), 595
- heap_caps_add_region (C++ *function*), 1139
- heap_caps_add_region_with_caps (C++ *function*), 1139
- heap_caps_aligned_alloc (C++ *function*), 1134
- heap_caps_aligned_calloc (C++ *function*), 1134
- heap_caps_aligned_free (C++ *function*), 1134
- heap_caps_calloc (C++ *function*), 1134
- heap_caps_calloc_prefer (C++ *function*), 1137
- heap_caps_check_integrity (C++ *function*), 1136
- heap_caps_check_integrity_addr (C++ *function*), 1136
- heap_caps_check_integrity_all (C++ *function*), 1136
- heap_caps_dump (C++ *function*), 1137
- heap_caps_dump_all (C++ *function*), 1137
- heap_caps_enable_nonos_stack_heaps (C++ *function*), 1139
- heap_caps_free (C++ *function*), 1134
- heap_caps_get_allocated_size (C++ *function*), 1137
- heap_caps_get_free_size (C++ *function*), 1135
- heap_caps_get_info (C++ *function*), 1135
- heap_caps_get_largest_free_block (C++ *function*), 1135
- heap_caps_get_minimum_free_size (C++ *function*), 1135
- heap_caps_get_total_size (C++ *function*), 1135
- heap_caps_init (C++ *function*), 1138
- heap_caps_malloc (C++ *function*), 1133
- heap_caps_malloc_extmem_enable (C++ *function*), 1136
- heap_caps_malloc_prefer (C++ *function*), 1136
- heap_caps_print_heap_info (C++ *function*), 1135

- heap_caps_realloc (C++ function), 1134
heap_caps_realloc_prefer (C++ function), 1137
HEAP_TRACE_ALL (C++ enumerator), 1152
heap_trace_dump (C++ function), 1152
heap_trace_get (C++ function), 1151
heap_trace_get_count (C++ function), 1151
heap_trace_init_standalone (C++ function), 1150
heap_trace_init_tohost (C++ function), 1151
HEAP_TRACE_LEAKS (C++ enumerator), 1152
heap_trace_mode_t (C++ enum), 1152
heap_trace_record_t (C++ class), 1152
heap_trace_record_t::address (C++ member), 1152
heap_trace_record_t::allocated_by (C++ member), 1152
heap_trace_record_t::ccount (C++ member), 1152
heap_trace_record_t::freed_by (C++ member), 1152
heap_trace_record_t::size (C++ member), 1152
heap_trace_resume (C++ function), 1151
heap_trace_start (C++ function), 1151
heap_trace_stop (C++ function), 1151
HTTP_AUTH_TYPE_BASIC (C++ enumerator), 831
HTTP_AUTH_TYPE_DIGEST (C++ enumerator), 831
HTTP_AUTH_TYPE_NONE (C++ enumerator), 831
HTTP_EVENT_DISCONNECTED (C++ enumerator), 830
HTTP_EVENT_ERROR (C++ enumerator), 830
http_event_handle_cb (C++ type), 830
HTTP_EVENT_HEADER_SENT (C++ enumerator), 830
HTTP_EVENT_HEADERS_SENT (C++ enumerator), 830
HTTP_EVENT_ON_CONNECTED (C++ enumerator), 830
HTTP_EVENT_ON_DATA (C++ enumerator), 830
HTTP_EVENT_ON_FINISH (C++ enumerator), 830
HTTP_EVENT_ON_HEADER (C++ enumerator), 830
HTTP_METHOD_DELETE (C++ enumerator), 831
HTTP_METHOD_GET (C++ enumerator), 831
HTTP_METHOD_HEAD (C++ enumerator), 831
HTTP_METHOD_MAX (C++ enumerator), 831
HTTP_METHOD_NOTIFY (C++ enumerator), 831
HTTP_METHOD_OPTIONS (C++ enumerator), 831
HTTP_METHOD_PATCH (C++ enumerator), 831
HTTP_METHOD_POST (C++ enumerator), 831
HTTP_METHOD_PUT (C++ enumerator), 831
HTTP_METHOD_SUBSCRIBE (C++ enumerator), 831
HTTP_METHOD_UNSUBSCRIBE (C++ enumerator), 831
HTTP_TRANSPORT_OVER_SSL (C++ enumerator), 830
HTTP_TRANSPORT_OVER_TCP (C++ enumerator), 830
HTTP_TRANSPORT_UNKNOWN (C++ enumerator), 830
HTTPD_200 (C macro), 849
HTTPD_204 (C macro), 849
HTTPD_207 (C macro), 849
HTTPD_400 (C macro), 849
HTTPD_400_BAD_REQUEST (C++ enumerator), 852
HTTPD_404 (C macro), 849
HTTPD_404_NOT_FOUND (C++ enumerator), 852
HTTPD_405_METHOD_NOT_ALLOWED (C++ enumerator), 852
HTTPD_408 (C macro), 849
HTTPD_408_REQ_TIMEOUT (C++ enumerator), 852
HTTPD_411_LENGTH_REQUIRED (C++ enumerator), 852
HTTPD_414_URI_TOO_LONG (C++ enumerator), 852
HTTPD_431_REQ_HDR_FIELDS_TOO_LARGE (C++ enumerator), 852
HTTPD_500 (C macro), 849
HTTPD_500_INTERNAL_SERVER_ERROR (C++ enumerator), 852
HTTPD_501_METHOD_NOT_IMPLEMENTED (C++ enumerator), 852
HTTPD_505_VERSION_NOT_SUPPORTED (C++ enumerator), 852
httpd_close_func_t (C++ type), 851
httpd_config (C++ class), 846
httpd_config::backlog_conn (C++ member), 846
httpd_config::close_fn (C++ member), 847
httpd_config::core_id (C++ member), 846
httpd_config::ctrl_port (C++ member), 846
httpd_config::global_transport_ctx (C++ member), 847
httpd_config::global_transport_ctx_free_fn (C++ member), 847
httpd_config::global_user_ctx (C++ member), 847
httpd_config::global_user_ctx_free_fn (C++ member), 847
httpd_config::lru_purge_enable (C++ member), 846
httpd_config::max_open_sockets (C++ member), 846
httpd_config::max_resp_headers (C++ member), 846
httpd_config::max_uri_handlers (C++ member), 846
httpd_config::open_fn (C++ member), 847
httpd_config::recv_wait_timeout (C++ member), 846
httpd_config::send_wait_timeout (C++ member), 847
httpd_config::server_port (C++ member), 846
httpd_config::stack_size (C++ member), 846

- [httpd_config::task_priority \(C++ member\), 846](#)
[httpd_config::uri_match_fn \(C++ member\), 847](#)
[httpd_config_t \(C++ type\), 852](#)
[HTTPD_DEFAULT_CONFIG \(C macro\), 849](#)
[HTTPD_ERR_CODE_MAX \(C++ enumerator\), 852](#)
[httpd_err_code_t \(C++ enum\), 852](#)
[httpd_err_handler_func_t \(C++ type\), 850](#)
[httpd_free_ctx_fn_t \(C++ type\), 851](#)
[httpd_get_global_transport_ctx \(C++ function\), 845](#)
[httpd_get_global_user_ctx \(C++ function\), 845](#)
[httpd_handle_t \(C++ type\), 851](#)
[HTTPD_MAX_REQ_HDR_LEN \(C macro\), 849](#)
[HTTPD_MAX_URI_LEN \(C macro\), 849](#)
[httpd_method_t \(C++ type\), 851](#)
[httpd_open_func_t \(C++ type\), 851](#)
[httpd_pending_func_t \(C++ type\), 850](#)
[httpd_query_key_value \(C++ function\), 838](#)
[httpd_queue_work \(C++ function\), 844](#)
[httpd_recv_func_t \(C++ type\), 850](#)
[httpd_register_err_handler \(C++ function\), 843](#)
[httpd_register_uri_handler \(C++ function\), 834](#)
[httpd_req \(C++ class\), 847](#)
[httpd_req::aux \(C++ member\), 848](#)
[httpd_req::content_len \(C++ member\), 848](#)
[httpd_req::free_ctx \(C++ member\), 848](#)
[httpd_req::handle \(C++ member\), 848](#)
[httpd_req::ignore_sess_ctx_changes \(C++ member\), 848](#)
[httpd_req::method \(C++ member\), 848](#)
[httpd_req::sess_ctx \(C++ member\), 848](#)
[httpd_req::uri \(C++ member\), 848](#)
[httpd_req::user_ctx \(C++ member\), 848](#)
[httpd_req_get_hdr_value_len \(C++ function\), 837](#)
[httpd_req_get_hdr_value_str \(C++ function\), 837](#)
[httpd_req_get_url_query_len \(C++ function\), 838](#)
[httpd_req_get_url_query_str \(C++ function\), 838](#)
[httpd_req_recv \(C++ function\), 836](#)
[httpd_req_t \(C++ type\), 850](#)
[httpd_req_to_sockfd \(C++ function\), 836](#)
[httpd_resp_send \(C++ function\), 839](#)
[httpd_resp_send_404 \(C++ function\), 842](#)
[httpd_resp_send_408 \(C++ function\), 842](#)
[httpd_resp_send_500 \(C++ function\), 842](#)
[httpd_resp_send_chunk \(C++ function\), 839](#)
[httpd_resp_send_err \(C++ function\), 841](#)
[httpd_resp_sendstr \(C++ function\), 840](#)
[httpd_resp_sendstr_chunk \(C++ function\), 840](#)
[httpd_resp_set_hdr \(C++ function\), 841](#)
[httpd_resp_set_status \(C++ function\), 840](#)
[httpd_resp_set_type \(C++ function\), 841](#)
[HTTPD_RESP_USE_STRLEN \(C macro\), 849](#)
[httpd_send \(C++ function\), 842](#)
[httpd_send_func_t \(C++ type\), 850](#)
[httpd_sess_get_ctx \(C++ function\), 844](#)
[httpd_sess_get_transport_ctx \(C++ function\), 845](#)
[httpd_sess_set_ctx \(C++ function\), 845](#)
[httpd_sess_set_pending_override \(C++ function\), 836](#)
[httpd_sess_set_recv_override \(C++ function\), 835](#)
[httpd_sess_set_send_override \(C++ function\), 836](#)
[httpd_sess_set_transport_ctx \(C++ function\), 845](#)
[httpd_sess_trigger_close \(C++ function\), 845](#)
[httpd_sess_update_lru_counter \(C++ function\), 846](#)
[HTTPD_SOCK_ERR_FAIL \(C macro\), 849](#)
[HTTPD_SOCK_ERR_INVALID \(C macro\), 849](#)
[HTTPD_SOCK_ERR_TIMEOUT \(C macro\), 849](#)
[httpd_ssl_config \(C++ class\), 853](#)
[httpd_ssl_config::cacert_len \(C++ member\), 854](#)
[httpd_ssl_config::cacert_pem \(C++ member\), 853](#)
[httpd_ssl_config::httpd \(C++ member\), 853](#)
[httpd_ssl_config::port_insecure \(C++ member\), 854](#)
[httpd_ssl_config::port_secure \(C++ member\), 854](#)
[httpd_ssl_config::prvtkey_len \(C++ member\), 854](#)
[httpd_ssl_config::prvtkey_pem \(C++ member\), 854](#)
[httpd_ssl_config::transport_mode \(C++ member\), 854](#)
[HTTPD_SSL_CONFIG_DEFAULT \(C macro\), 854](#)
[httpd_ssl_config_t \(C++ type\), 854](#)
[httpd_ssl_start \(C++ function\), 853](#)
[httpd_ssl_stop \(C++ function\), 853](#)
[HTTPD_SSL_TRANSPORT_INSECURE \(C++ enumerator\), 854](#)
[httpd_ssl_transport_mode_t \(C++ enum\), 854](#)
[HTTPD_SSL_TRANSPORT_SECURE \(C++ enumerator\), 854](#)
[httpd_start \(C++ function\), 843](#)
[httpd_stop \(C++ function\), 844](#)
[HTTPD_TYPE_JSON \(C macro\), 849](#)
[HTTPD_TYPE_OCTET \(C macro\), 849](#)
[HTTPD_TYPE_TEXT \(C macro\), 849](#)
[httpd_unregister_uri \(C++ function\), 835](#)

- [httpd_unregister_uri_handler \(C++ function\), 835](#)
[httpd_uri \(C++ class\), 848](#)
[httpd_uri::handler \(C++ member\), 848](#)
[httpd_uri::method \(C++ member\), 848](#)
[httpd_uri::uri \(C++ member\), 848](#)
[httpd_uri::user_ctx \(C++ member\), 848](#)
[httpd_uri_match_func_t \(C++ type\), 851](#)
[httpd_uri_match_wildcard \(C++ function\), 838](#)
[httpd_uri_t \(C++ type\), 850](#)
[httpd_work_fn_t \(C++ type\), 852](#)
[HttpStatus_Code \(C++ enum\), 831](#)
[HttpStatus_Found \(C++ enumerator\), 831](#)
[HttpStatus_MovedPermanently \(C++ enumerator\), 831](#)
[HttpStatus_MultipleChoices \(C++ enumerator\), 831](#)
[HttpStatus_Ok \(C++ enumerator\), 831](#)
[HttpStatus_TemporaryRedirect \(C++ enumerator\), 831](#)
[HttpStatus_Unauthorized \(C++ enumerator\), 831](#)
- I**
- [i2c_ack_type_t \(C++ enum\), 646](#)
[I2C_ADDR_BIT_10 \(C++ enumerator\), 646](#)
[I2C_ADDR_BIT_7 \(C++ enumerator\), 646](#)
[I2C_ADDR_BIT_MAX \(C++ enumerator\), 646](#)
[i2c_addr_mode_t \(C++ enum\), 645](#)
[I2C_APB_CLK_FREQ \(C macro\), 644](#)
[I2C_CMD_END \(C++ enumerator\), 645](#)
[i2c_cmd_handle_t \(C++ type\), 644](#)
[i2c_cmd_link_create \(C++ function\), 639](#)
[i2c_cmd_link_delete \(C++ function\), 639](#)
[I2C_CMD_READ \(C++ enumerator\), 645](#)
[I2C_CMD_RESTART \(C++ enumerator\), 645](#)
[I2C_CMD_STOP \(C++ enumerator\), 645](#)
[I2C_CMD_WRITE \(C++ enumerator\), 645](#)
[i2c_config_t \(C++ class\), 644](#)
[i2c_config_t::addr_10bit_en \(C++ member\), 645](#)
[i2c_config_t::clk_speed \(C++ member\), 645](#)
[i2c_config_t::mode \(C++ member\), 644](#)
[i2c_config_t::scl_io_num \(C++ member\), 644](#)
[i2c_config_t::scl_pullup_en \(C++ member\), 644](#)
[i2c_config_t::sda_io_num \(C++ member\), 644](#)
[i2c_config_t::sda_pullup_en \(C++ member\), 644](#)
[i2c_config_t::slave_addr \(C++ member\), 645](#)
[I2C_DATA_MODE_LSB_FIRST \(C++ enumerator\), 645](#)
[I2C_DATA_MODE_MAX \(C++ enumerator\), 645](#)
[I2C_DATA_MODE_MSB_FIRST \(C++ enumerator\), 645](#)
[i2c_driver_delete \(C++ function\), 638](#)
[i2c_driver_install \(C++ function\), 638](#)
[i2c_filter_disable \(C++ function\), 642](#)
[i2c_filter_enable \(C++ function\), 642](#)
[i2c_get_data_mode \(C++ function\), 644](#)
[i2c_get_data_timing \(C++ function\), 643](#)
[i2c_get_period \(C++ function\), 642](#)
[i2c_get_start_timing \(C++ function\), 642](#)
[i2c_get_stop_timing \(C++ function\), 643](#)
[i2c_get_timeout \(C++ function\), 643](#)
[i2c_isr_free \(C++ function\), 639](#)
[i2c_isr_register \(C++ function\), 638](#)
[I2C_MASTER_ACK \(C++ enumerator\), 646](#)
[I2C_MASTER_ACK_MAX \(C++ enumerator\), 646](#)
[i2c_master_cmd_begin \(C++ function\), 641](#)
[I2C_MASTER_LAST_NACK \(C++ enumerator\), 646](#)
[I2C_MASTER_NACK \(C++ enumerator\), 646](#)
[I2C_MASTER_READ \(C++ enumerator\), 645](#)
[i2c_master_read \(C++ function\), 640](#)
[i2c_master_read_byte \(C++ function\), 640](#)
[i2c_master_start \(C++ function\), 639](#)
[i2c_master_stop \(C++ function\), 640](#)
[I2C_MASTER_WRITE \(C++ enumerator\), 645](#)
[i2c_master_write \(C++ function\), 640](#)
[i2c_master_write_byte \(C++ function\), 639](#)
[I2C_MODE_MASTER \(C++ enumerator\), 645](#)
[I2C_MODE_MAX \(C++ enumerator\), 645](#)
[I2C_MODE_SLAVE \(C++ enumerator\), 645](#)
[i2c_mode_t \(C++ enum\), 645](#)
[I2C_NUM_0 \(C macro\), 644](#)
[I2C_NUM_1 \(C macro\), 644](#)
[I2C_NUM_MAX \(C macro\), 644](#)
[i2c_opmode_t \(C++ enum\), 645](#)
[i2c_param_config \(C++ function\), 638](#)
[i2c_port_t \(C++ type\), 645](#)
[i2c_reset_rx_fifo \(C++ function\), 638](#)
[i2c_reset_tx_fifo \(C++ function\), 638](#)
[i2c_rw_t \(C++ enum\), 645](#)
[I2C_SCLK_APB \(C++ enumerator\), 646](#)
[I2C_SCLK_REF_TICK \(C++ enumerator\), 646](#)
[i2c_sclk_t \(C++ enum\), 646](#)
[i2c_set_data_mode \(C++ function\), 643](#)
[i2c_set_data_timing \(C++ function\), 643](#)
[i2c_set_period \(C++ function\), 641](#)
[i2c_set_pin \(C++ function\), 639](#)
[i2c_set_start_timing \(C++ function\), 642](#)
[i2c_set_stop_timing \(C++ function\), 642](#)
[i2c_set_timeout \(C++ function\), 643](#)
[i2c_slave_read_buffer \(C++ function\), 641](#)
[i2c_slave_write_buffer \(C++ function\), 641](#)
[i2c_trans_mode_t \(C++ enum\), 645](#)
[i2s_adc_disable \(C++ function\), 653](#)
[i2s_adc_enable \(C++ function\), 653](#)
[I2S_BITS_PER_SAMPLE_16BIT \(C++ enumerator\), 654](#)

- I2S_BITS_PER_SAMPLE_24BIT (C++ *enumerator*), 654
- I2S_BITS_PER_SAMPLE_32BIT (C++ *enumerator*), 655
- I2S_BITS_PER_SAMPLE_8BIT (C++ *enumerator*), 654
- i2s_bits_per_sample_t (C++ *enum*), 654
- I2S_CHANNEL_FMT_ALL_LEFT (C++ *enumerator*), 655
- I2S_CHANNEL_FMT_ALL_RIGHT (C++ *enumerator*), 655
- I2S_CHANNEL_FMT_ONLY_LEFT (C++ *enumerator*), 655
- I2S_CHANNEL_FMT_ONLY_RIGHT (C++ *enumerator*), 655
- I2S_CHANNEL_FMT_RIGHT_LEFT (C++ *enumerator*), 655
- i2s_channel_fmt_t (C++ *enum*), 655
- I2S_CHANNEL_MONO (C++ *enumerator*), 655
- I2S_CHANNEL_STEREO (C++ *enumerator*), 655
- i2s_channel_t (C++ *enum*), 655
- I2S_CLK_APLL (C++ *enumerator*), 656
- I2S_CLK_D2CLK (C++ *enumerator*), 656
- i2s_clock_src_t (C++ *enum*), 656
- I2S_COMM_FORMAT_I2S (C++ *enumerator*), 655
- I2S_COMM_FORMAT_I2S_LSB (C++ *enumerator*), 655
- I2S_COMM_FORMAT_I2S_MSB (C++ *enumerator*), 655
- I2S_COMM_FORMAT_PCM (C++ *enumerator*), 655
- I2S_COMM_FORMAT_PCM_LONG (C++ *enumerator*), 655
- I2S_COMM_FORMAT_PCM_SHORT (C++ *enumerator*), 655
- i2s_comm_format_t (C++ *enum*), 655
- i2s_config_t (C++ *class*), 653
- i2s_config_t::bits_per_sample (C++ *member*), 653
- i2s_config_t::channel_format (C++ *member*), 653
- i2s_config_t::communication_format (C++ *member*), 653
- i2s_config_t::dma_buf_count (C++ *member*), 653
- i2s_config_t::dma_buf_len (C++ *member*), 653
- i2s_config_t::fixed_mclk (C++ *member*), 654
- i2s_config_t::intr_alloc_flags (C++ *member*), 653
- i2s_config_t::mode (C++ *member*), 653
- i2s_config_t::sample_rate (C++ *member*), 653
- i2s_config_t::tx_desc_auto_clear (C++ *member*), 654
- i2s_config_t::use_apll (C++ *member*), 654
- I2S_DAC_CHANNEL_BOTH_EN (C++ *enumerator*), 656
- I2S_DAC_CHANNEL_DISABLE (C++ *enumerator*), 656
- I2S_DAC_CHANNEL_LEFT_EN (C++ *enumerator*), 656
- I2S_DAC_CHANNEL_MAX (C++ *enumerator*), 656
- I2S_DAC_CHANNEL_RIGHT_EN (C++ *enumerator*), 656
- i2s_dac_mode_t (C++ *enum*), 656
- i2s_driver_install (C++ *function*), 650
- i2s_driver_uninstall (C++ *function*), 650
- I2S_EVENT_DMA_ERROR (C++ *enumerator*), 656
- I2S_EVENT_MAX (C++ *enumerator*), 656
- I2S_EVENT_RX_DONE (C++ *enumerator*), 656
- i2s_event_t (C++ *class*), 654
- i2s_event_t::size (C++ *member*), 654
- i2s_event_t::type (C++ *member*), 654
- I2S_EVENT_TX_DONE (C++ *enumerator*), 656
- i2s_event_type_t (C++ *enum*), 656
- i2s_get_clk (C++ *function*), 652
- i2s_isr_handle_t (C++ *type*), 653
- I2S_MODE_ADC_BUILT_IN (C++ *enumerator*), 656
- I2S_MODE_DAC_BUILT_IN (C++ *enumerator*), 656
- I2S_MODE_MASTER (C++ *enumerator*), 656
- I2S_MODE_RX (C++ *enumerator*), 656
- I2S_MODE_SLAVE (C++ *enumerator*), 656
- i2s_mode_t (C++ *enum*), 655
- I2S_MODE_TX (C++ *enumerator*), 656
- i2s_pin_config_t (C++ *class*), 654
- i2s_pin_config_t::bck_io_num (C++ *member*), 654
- i2s_pin_config_t::data_in_num (C++ *member*), 654
- i2s_pin_config_t::data_out_num (C++ *member*), 654
- i2s_pin_config_t::ws_io_num (C++ *member*), 654
- I2S_PIN_NO_CHANGE (C *macro*), 654
- i2s_port_t (C++ *type*), 654
- i2s_read (C++ *function*), 651
- i2s_set_adc_mode (C++ *function*), 652
- i2s_set_clk (C++ *function*), 652
- i2s_set_dac_mode (C++ *function*), 650
- i2s_set_pin (C++ *function*), 649
- i2s_set_sample_rates (C++ *function*), 651
- i2s_start (C++ *function*), 652
- i2s_stop (C++ *function*), 651
- i2s_write (C++ *function*), 650
- i2s_write_expand (C++ *function*), 651
- i2s_zero_dma_buffer (C++ *function*), 652
- I_ADDI (C *macro*), 1672
- I_ADDR (C *macro*), 1671
- I_ANDI (C *macro*), 1672
- I_ANDR (C *macro*), 1671
- I_BGE (C *macro*), 1671
- I_BL (C *macro*), 1671
- I_BXFI (C *macro*), 1671
- I_BXFR (C *macro*), 1671
- I_BXI (C *macro*), 1671

- [I_BXR \(C macro\), 1671](#)
[I_BXZI \(C macro\), 1671](#)
[I_BXZR \(C macro\), 1671](#)
[I_DELAY \(C macro\), 1670](#)
[I_END \(C macro\), 1670](#)
[I_HALT \(C macro\), 1670](#)
[I_LD \(C macro\), 1671](#)
[I_LSHI \(C macro\), 1672](#)
[I_LSHR \(C macro\), 1672](#)
[I_MOVI \(C macro\), 1672](#)
[I_MOVR \(C macro\), 1671](#)
[I_ORI \(C macro\), 1672](#)
[I_ORR \(C macro\), 1671](#)
[I_RD_REG \(C macro\), 1671](#)
[I_RSHI \(C macro\), 1672](#)
[I_RSHR \(C macro\), 1672](#)
[I_ST \(C macro\), 1670](#)
[I_SUBI \(C macro\), 1672](#)
[I_SUBR \(C macro\), 1671](#)
[I_WR_REG \(C macro\), 1671](#)
[intr_handle_data_t \(C++ type\), 1166](#)
[intr_handle_t \(C++ type\), 1166](#)
[intr_handler_t \(C++ type\), 1166](#)
- ## L
- [LEDC_APB_CLK \(C++ enumerator\), 667](#)
[LEDC_APB_CLK_HZ \(C macro\), 665](#)
[LEDC_AUTO_CLK \(C++ enumerator\), 667](#)
[ledc_bind_channel_timer \(C++ function\), 663](#)
[LEDC_CHANNEL_0 \(C++ enumerator\), 668](#)
[LEDC_CHANNEL_1 \(C++ enumerator\), 668](#)
[LEDC_CHANNEL_2 \(C++ enumerator\), 668](#)
[LEDC_CHANNEL_3 \(C++ enumerator\), 668](#)
[LEDC_CHANNEL_4 \(C++ enumerator\), 668](#)
[LEDC_CHANNEL_5 \(C++ enumerator\), 668](#)
[LEDC_CHANNEL_6 \(C++ enumerator\), 668](#)
[LEDC_CHANNEL_7 \(C++ enumerator\), 668](#)
[ledc_channel_config \(C++ function\), 660](#)
[ledc_channel_config_t \(C++ class\), 666](#)
[ledc_channel_config_t::channel \(C++ member\), 666](#)
[ledc_channel_config_t::duty \(C++ member\), 666](#)
[ledc_channel_config_t::gpio_num \(C++ member\), 666](#)
[ledc_channel_config_t::hpoint \(C++ member\), 666](#)
[ledc_channel_config_t::intr_type \(C++ member\), 666](#)
[ledc_channel_config_t::speed_mode \(C++ member\), 666](#)
[ledc_channel_config_t::timer_sel \(C++ member\), 666](#)
[LEDC_CHANNEL_MAX \(C++ enumerator\), 668](#)
[ledc_channel_t \(C++ enum\), 668](#)
[ledc_clk_cfg_t \(C++ enum\), 667](#)
[ledc_clk_src_t \(C++ enum\), 667](#)
[LEDC_DUTY_DIR_DECREASE \(C++ enumerator\), 667](#)
[LEDC_DUTY_DIR_INCREASE \(C++ enumerator\), 667](#)
[LEDC_DUTY_DIR_MAX \(C++ enumerator\), 667](#)
[ledc_duty_direction_t \(C++ enum\), 667](#)
[LEDC_ERR_DUTY \(C macro\), 665](#)
[LEDC_ERR_VAL \(C macro\), 665](#)
[ledc_fade_func_install \(C++ function\), 664](#)
[ledc_fade_func_uninstall \(C++ function\), 664](#)
[LEDC_FADE_MAX \(C++ enumerator\), 669](#)
[ledc_fade_mode_t \(C++ enum\), 669](#)
[LEDC_FADE_NO_WAIT \(C++ enumerator\), 669](#)
[ledc_fade_start \(C++ function\), 664](#)
[LEDC_FADE_WAIT_DONE \(C++ enumerator\), 669](#)
[ledc_get_duty \(C++ function\), 662](#)
[ledc_get_freq \(C++ function\), 661](#)
[ledc_get_hpoint \(C++ function\), 661](#)
[LEDC_INTR_DISABLE \(C++ enumerator\), 667](#)
[LEDC_INTR_FADE_END \(C++ enumerator\), 667](#)
[LEDC_INTR_MAX \(C++ enumerator\), 667](#)
[ledc_intr_type_t \(C++ enum\), 667](#)
[ledc_isr_handle_t \(C++ type\), 666](#)
[ledc_isr_register \(C++ function\), 662](#)
[LEDC_LOW_SPEED_MODE \(C++ enumerator\), 667](#)
[ledc_mode_t \(C++ enum\), 667](#)
[LEDC_REF_CLK_HZ \(C macro\), 665](#)
[LEDC_REF_TICK \(C++ enumerator\), 667](#)
[ledc_set_duty \(C++ function\), 661](#)
[ledc_set_duty_and_update \(C++ function\), 665](#)
[ledc_set_duty_with_hpoint \(C++ function\), 661](#)
[ledc_set_fade \(C++ function\), 662](#)
[ledc_set_fade_step_and_start \(C++ function\), 665](#)
[ledc_set_fade_time_and_start \(C++ function\), 665](#)
[ledc_set_fade_with_step \(C++ function\), 663](#)
[ledc_set_fade_with_time \(C++ function\), 664](#)
[ledc_set_freq \(C++ function\), 661](#)
[ledc_set_pin \(C++ function\), 660](#)
[LEDC_SLOW_CLK_APB \(C++ enumerator\), 667](#)
[LEDC_SLOW_CLK_RTC8M \(C++ enumerator\), 667](#)
[ledc_slow_clk_sel_t \(C++ enum\), 667](#)
[LEDC_SPEED_MODE_MAX \(C++ enumerator\), 667](#)
[ledc_stop \(C++ function\), 660](#)
[LEDC_TIMER_0 \(C++ enumerator\), 667](#)
[LEDC_TIMER_1 \(C++ enumerator\), 668](#)
[LEDC_TIMER_10_BIT \(C++ enumerator\), 669](#)
[LEDC_TIMER_11_BIT \(C++ enumerator\), 669](#)
[LEDC_TIMER_12_BIT \(C++ enumerator\), 669](#)
[LEDC_TIMER_13_BIT \(C++ enumerator\), 669](#)
[LEDC_TIMER_14_BIT \(C++ enumerator\), 669](#)
[LEDC_TIMER_15_BIT \(C++ enumerator\), 669](#)
[LEDC_TIMER_16_BIT \(C++ enumerator\), 669](#)
[LEDC_TIMER_17_BIT \(C++ enumerator\), 669](#)

- LEDC_TIMER_18_BIT (C++ enumerator), 669
LEDC_TIMER_19_BIT (C++ enumerator), 669
LEDC_TIMER_1_BIT (C++ enumerator), 668
LEDC_TIMER_2 (C++ enumerator), 668
LEDC_TIMER_20_BIT (C++ enumerator), 669
LEDC_TIMER_2_BIT (C++ enumerator), 668
LEDC_TIMER_3 (C++ enumerator), 668
LEDC_TIMER_3_BIT (C++ enumerator), 668
LEDC_TIMER_4_BIT (C++ enumerator), 668
LEDC_TIMER_5_BIT (C++ enumerator), 668
LEDC_TIMER_6_BIT (C++ enumerator), 668
LEDC_TIMER_7_BIT (C++ enumerator), 668
LEDC_TIMER_8_BIT (C++ enumerator), 668
LEDC_TIMER_9_BIT (C++ enumerator), 668
LEDC_TIMER_BIT_MAX (C++ enumerator), 669
ledc_timer_bit_t (C++ enum), 668
ledc_timer_config (C++ function), 660
ledc_timer_config_t (C++ class), 666
ledc_timer_config_t::bit_num (C++ member), 666
ledc_timer_config_t::clk_cfg (C++ member), 666
ledc_timer_config_t::duty_resolution (C++ member), 666
ledc_timer_config_t::freq_hz (C++ member), 666
ledc_timer_config_t::speed_mode (C++ member), 666
ledc_timer_config_t::timer_num (C++ member), 666
LEDC_TIMER_MAX (C++ enumerator), 668
ledc_timer_pause (C++ function), 663
ledc_timer_resume (C++ function), 663
ledc_timer_rst (C++ function), 663
ledc_timer_set (C++ function), 662
ledc_timer_t (C++ enum), 667
ledc_update_duty (C++ function), 660
LEDC_USE_APB_CLK (C++ enumerator), 667
LEDC_USE_REF_TICK (C++ enumerator), 667
LEDC_USE_RTC8M_CLK (C++ enumerator), 667
- ## M
- M_BGE (C macro), 1672
M_BL (C macro), 1672
M_BX (C macro), 1672
M_BXF (C macro), 1672
M_BXZ (C macro), 1672
M_LABEL (C macro), 1672
MALLOC_CAP_32BIT (C macro), 1137
MALLOC_CAP_8BIT (C macro), 1137
MALLOC_CAP_DEFAULT (C macro), 1138
MALLOC_CAP_DMA (C macro), 1137
MALLOC_CAP_EXEC (C macro), 1137
MALLOC_CAP_INTERNAL (C macro), 1138
MALLOC_CAP_INVALID (C macro), 1138
MALLOC_CAP_PID2 (C macro), 1138
MALLOC_CAP_PID3 (C macro), 1138
MALLOC_CAP_PID4 (C macro), 1138
MALLOC_CAP_PID5 (C macro), 1138
MALLOC_CAP_PID6 (C macro), 1138
MALLOC_CAP_PID7 (C macro), 1138
MALLOC_CAP_SPIRAM (C macro), 1138
MAX_BLE_DEVNAME_LEN (C macro), 899
MAX_FDS (C macro), 992
MAX_PASSPHRASE_LEN (C macro), 515
MAX_SSID_LEN (C macro), 515
MAX_WPS_AP_CRED (C macro), 515
mbc_master_destroy (C++ function), 876
mbc_master_get_cid_info (C++ function), 878
mbc_master_get_parameter (C++ function), 878
mbc_master_init (C++ function), 875
mbc_master_send_request (C++ function), 877
mbc_master_set_descriptor (C++ function), 877
mbc_master_set_parameter (C++ function), 878
mbc_master_setup (C++ function), 875
mbc_master_start (C++ function), 876
mbc_slave_check_event (C++ function), 876
mbc_slave_destroy (C++ function), 876
mbc_slave_get_param_info (C++ function), 877
mbc_slave_init (C++ function), 875
mbc_slave_set_descriptor (C++ function), 876
mbc_slave_setup (C++ function), 875
mbc_slave_start (C++ function), 876
MCPWM0A (C++ enumerator), 685
MCPWM0B (C++ enumerator), 685
MCPWM1A (C++ enumerator), 685
MCPWM1B (C++ enumerator), 685
MCPWM2A (C++ enumerator), 685
MCPWM2B (C++ enumerator), 685
MCPWM_ACTION_FORCE_HIGH (C++ enumerator), 675
MCPWM_ACTION_FORCE_LOW (C++ enumerator), 675
MCPWM_ACTION_NO_CHANGE (C++ enumerator), 675
mcpwm_action_on_pwmxa_t (C++ type), 685
mcpwm_action_on_pwmxb_t (C++ type), 685
MCPWM_ACTION_TOGGLE (C++ enumerator), 675
MCPWM_ACTIVE_HIGH_COMPLIMENT_MODE (C++ enumerator), 676
MCPWM_ACTIVE_HIGH_MODE (C++ enumerator), 676
MCPWM_ACTIVE_LOW_COMPLIMENT_MODE (C++ enumerator), 676
MCPWM_ACTIVE_LOW_MODE (C++ enumerator), 676
MCPWM_ACTIVE_RED_FED_FROM_PWMXA (C++ enumerator), 676
MCPWM_ACTIVE_RED_FED_FROM_PWMXB (C++ enumerator), 676
MCPWM_BYPASS_FED (C++ enumerator), 676
MCPWM_BYPASS_RED (C++ enumerator), 676

- MCPWM_CAP_0 (C++ enumerator), 686
- MCPWM_CAP_1 (C++ enumerator), 686
- MCPWM_CAP_2 (C++ enumerator), 686
- mcpwm_capture_disable (C++ function), 682
- mcpwm_capture_enable (C++ function), 682
- mcpwm_capture_on_edge_t (C++ enum), 676
- mcpwm_capture_signal_get_edge (C++ function), 682
- mcpwm_capture_signal_get_value (C++ function), 682
- mcpwm_capture_signal_t (C++ enum), 687
- mcpwm_carrier_config_t (C++ class), 684
- mcpwm_carrier_config_t::carrier_duty (C++ member), 684
- mcpwm_carrier_config_t::carrier_ivt_mode (C++ member), 684
- mcpwm_carrier_config_t::carrier_os_mode (C++ member), 684
- mcpwm_carrier_config_t::carrier_period (C++ member), 684
- mcpwm_carrier_config_t::pulse_width_in_ns (C++ member), 684
- mcpwm_carrier_disable (C++ function), 679
- mcpwm_carrier_enable (C++ function), 679
- mcpwm_carrier_init (C++ function), 679
- mcpwm_carrier_oneshot_mode_disable (C++ function), 680
- mcpwm_carrier_oneshot_mode_enable (C++ function), 680
- mcpwm_carrier_os_t (C++ enum), 686
- MCPWM_CARRIER_OUT_IVT_DIS (C++ enumerator), 687
- MCPWM_CARRIER_OUT_IVT_EN (C++ enumerator), 687
- mcpwm_carrier_out_ivt_t (C++ enum), 686
- mcpwm_carrier_output_invert (C++ function), 680
- mcpwm_carrier_set_duty_cycle (C++ function), 680
- mcpwm_carrier_set_period (C++ function), 679
- mcpwm_config_t (C++ class), 684
- mcpwm_config_t::cmpr_a (C++ member), 684
- mcpwm_config_t::cmpr_b (C++ member), 684
- mcpwm_config_t::counter_mode (C++ member), 684
- mcpwm_config_t::duty_mode (C++ member), 684
- mcpwm_config_t::frequency (C++ member), 684
- MCPWM_COUNTER_MAX (C++ enumerator), 675
- mcpwm_counter_type_t (C++ enum), 675
- MCPWM_DEADTIME_BYPASS (C++ enumerator), 676
- mcpwm_deadtime_disable (C++ function), 681
- mcpwm_deadtime_enable (C++ function), 680
- MCPWM_DEADTIME_TYPE_MAX (C++ enumerator), 676
- mcpwm_deadtime_type_t (C++ enum), 675
- MCPWM_DOWN_COUNTER (C++ enumerator), 675
- MCPWM_DUTY_MODE_0 (C++ enumerator), 675
- MCPWM_DUTY_MODE_1 (C++ enumerator), 675
- MCPWM_DUTY_MODE_MAX (C++ enumerator), 675
- mcpwm_duty_type_t (C++ enum), 675
- MCPWM_FAULT_0 (C++ enumerator), 685
- MCPWM_FAULT_1 (C++ enumerator), 685
- MCPWM_FAULT_2 (C++ enumerator), 686
- mcpwm_fault_deinit (C++ function), 682
- mcpwm_fault_init (C++ function), 681
- mcpwm_fault_input_level_t (C++ enum), 687
- mcpwm_fault_set_cyc_mode (C++ function), 681
- mcpwm_fault_set_oneshot_mode (C++ function), 681
- mcpwm_fault_signal_t (C++ enum), 687
- MCPWM_FORCE_MCPWMXA_HIGH (C macro), 685
- MCPWM_FORCE_MCPWMXA_LOW (C macro), 685
- MCPWM_FORCE_MCPWXB_HIGH (C macro), 685
- MCPWM_FORCE_MCPWXB_LOW (C macro), 685
- MCPWM_GEN_A (C++ enumerator), 686
- MCPWM_GEN_B (C++ enumerator), 686
- MCPWM_GEN_MAX (C++ enumerator), 686
- mcpwm_generator_t (C++ enum), 686
- mcpwm_get_duty (C++ function), 678
- mcpwm_get_frequency (C++ function), 678
- mcpwm_gpio_init (C++ function), 677
- MCPWM_HAL_GENERATOR_MODE_FORCE_HIGH (C++ enumerator), 675
- MCPWM_HAL_GENERATOR_MODE_FORCE_LOW (C++ enumerator), 675
- MCPWM_HIGH_LEVEL_TGR (C++ enumerator), 687
- mcpwm_init (C++ function), 677
- mcpwm_intr_t (C++ enum), 675
- mcpwm_io_signals_t (C++ enum), 685
- mcpwm_isr_register (C++ function), 683
- MCPWM_LL_INTR_CAP0 (C++ enumerator), 675
- MCPWM_LL_INTR_CAP1 (C++ enumerator), 675
- MCPWM_LL_INTR_CAP2 (C++ enumerator), 675
- MCPWM_LOW_LEVEL_TGR (C++ enumerator), 687
- MCPWM_NEG_EDGE (C++ enumerator), 676
- MCPWM_NO_CHANGE_IN_MCPWMXA (C macro), 685
- MCPWM_NO_CHANGE_IN_MCPWXB (C macro), 685
- MCPWM_ONESHOT_MODE_DIS (C++ enumerator), 686
- MCPWM_ONESHOT_MODE_EN (C++ enumerator), 686
- mcpwm_operator_t (C++ type), 685
- MCPWM_OPR_A (C macro), 685
- MCPWM_OPR_B (C macro), 685
- MCPWM_OPR_MAX (C macro), 685
- mcpwm_output_action_t (C++ enum), 675
- mcpwm_pin_config_t (C++ class), 683
- mcpwm_pin_config_t::mcpwm0a_out_num (C++ member), 683
- mcpwm_pin_config_t::mcpwm0b_out_num (C++ member), 683
- mcpwm_pin_config_t::mcpwm1a_out_num (C++ member), 683

- mcpwm_pin_config_t::mcpwm1b_out_num (C++ member), 683
 mcpwm_pin_config_t::mcpwm2a_out_num (C++ member), 683
 mcpwm_pin_config_t::mcpwm2b_out_num (C++ member), 683
 mcpwm_pin_config_t::mcpwm_cap0_in_num (C++ member), 684
 mcpwm_pin_config_t::mcpwm_cap1_in_num (C++ member), 684
 mcpwm_pin_config_t::mcpwm_cap2_in_num (C++ member), 684
 mcpwm_pin_config_t::mcpwm_fault0_in_num (C++ member), 684
 mcpwm_pin_config_t::mcpwm_fault1_in_num (C++ member), 684
 mcpwm_pin_config_t::mcpwm_fault2_in_num (C++ member), 684
 mcpwm_pin_config_t::mcpwm_sync0_in_num (C++ member), 683
 mcpwm_pin_config_t::mcpwm_sync1_in_num (C++ member), 683
 mcpwm_pin_config_t::mcpwm_sync2_in_num (C++ member), 683
 MCPWM_POS_EDGE (C++ enumerator), 676
 MCPWM_SELECT_CAP0 (C++ enumerator), 687
 MCPWM_SELECT_CAP1 (C++ enumerator), 687
 MCPWM_SELECT_CAP2 (C++ enumerator), 687
 MCPWM_SELECT_F0 (C++ enumerator), 687
 MCPWM_SELECT_F1 (C++ enumerator), 687
 MCPWM_SELECT_F2 (C++ enumerator), 687
 MCPWM_SELECT_SYNC0 (C++ enumerator), 676
 MCPWM_SELECT_SYNC1 (C++ enumerator), 676
 MCPWM_SELECT_SYNC2 (C++ enumerator), 676
 mcpwm_set_duty (C++ function), 677
 mcpwm_set_duty_in_us (C++ function), 677
 mcpwm_set_duty_type (C++ function), 678
 mcpwm_set_frequency (C++ function), 677
 mcpwm_set_pin (C++ function), 677
 mcpwm_set_signal_high (C++ function), 678
 mcpwm_set_signal_low (C++ function), 678
 mcpwm_start (C++ function), 679
 mcpwm_stop (C++ function), 679
 MCPWM_SYNC_0 (C++ enumerator), 685
 MCPWM_SYNC_1 (C++ enumerator), 685
 MCPWM_SYNC_2 (C++ enumerator), 685
 mcpwm_sync_disable (C++ function), 683
 mcpwm_sync_enable (C++ function), 682
 mcpwm_sync_signal_t (C++ enum), 676
 MCPWM_TIMER_0 (C++ enumerator), 686
 MCPWM_TIMER_1 (C++ enumerator), 686
 MCPWM_TIMER_2 (C++ enumerator), 686
 MCPWM_TIMER_MAX (C++ enumerator), 686
 mcpwm_timer_t (C++ enum), 686
 MCPWM_TOG_MCPWMXA (C macro), 685
 MCPWM_TOG_MCPWMXB (C macro), 685
 MCPWM_UNIT_0 (C++ enumerator), 686
 MCPWM_UNIT_1 (C++ enumerator), 686
 MCPWM_UNIT_MAX (C++ enumerator), 686
 mcpwm_unit_t (C++ enum), 686
 MCPWM_UP_COUNTER (C++ enumerator), 675
 MCPWM_UP_DOWN_COUNTER (C++ enumerator), 675
 mdns_free (C++ function), 869
 mdns_handle_system_event (C++ function), 873
 mdns_hostname_set (C++ function), 869
 MDNS_IF_AP (C++ enumerator), 874
 MDNS_IF_ETH (C++ enumerator), 874
 mdns_if_internal (C++ enum), 874
 MDNS_IF_MAX (C++ enumerator), 874
 MDNS_IF_STA (C++ enumerator), 874
 mdns_if_t (C++ type), 874
 mdns_init (C++ function), 869
 mdns_instance_name_set (C++ function), 869
 mdns_ip_addr_s (C++ class), 873
 mdns_ip_addr_s::addr (C++ member), 873
 mdns_ip_addr_s::next (C++ member), 873
 mdns_ip_addr_t (C++ type), 874
 MDNS_IP_PROTOCOL_MAX (C++ enumerator), 874
 mdns_ip_protocol_t (C++ enum), 874
 MDNS_IP_PROTOCOL_V4 (C++ enumerator), 874
 MDNS_IP_PROTOCOL_V6 (C++ enumerator), 874
 mdns_query (C++ function), 871
 mdns_query_a (C++ function), 872
 mdns_query_aaaa (C++ function), 872
 mdns_query_ptr (C++ function), 871
 mdns_query_results_free (C++ function), 871
 mdns_query_srv (C++ function), 872
 mdns_query_txt (C++ function), 872
 mdns_result_s (C++ class), 873
 mdns_result_s::addr (C++ member), 874
 mdns_result_s::hostname (C++ member), 873
 mdns_result_s::instance_name (C++ member), 873
 mdns_result_s::ip_protocol (C++ member), 873
 mdns_result_s::next (C++ member), 873
 mdns_result_s::port (C++ member), 873
 mdns_result_s::tcpip_if (C++ member), 873
 mdns_result_s::txt (C++ member), 874
 mdns_result_s::txt_count (C++ member), 874
 mdns_result_t (C++ type), 874
 mdns_service_add (C++ function), 869
 mdns_service_instance_name_set (C++ function), 870
 mdns_service_port_set (C++ function), 870
 mdns_service_remove (C++ function), 870
 mdns_service_remove_all (C++ function), 871
 mdns_service_txt_item_remove (C++ function), 871
 mdns_service_txt_item_set (C++ function), 871
 mdns_service_txt_set (C++ function), 870
 mdns_txt_item_t (C++ class), 873
 mdns_txt_item_t::key (C++ member), 873

- mdns_txt_item_t::value (C++ member), 873
- MDNS_TYPE_A (C macro), 874
- MDNS_TYPE_AAAA (C macro), 874
- MDNS_TYPE_ANY (C macro), 874
- MDNS_TYPE_NSEC (C macro), 874
- MDNS_TYPE_OPT (C macro), 874
- MDNS_TYPE_PTR (C macro), 874
- MDNS_TYPE_SRV (C macro), 874
- MDNS_TYPE_TXT (C macro), 874
- mesh_addr_t (C++ union), 548
- mesh_addr_t::addr (C++ member), 548
- mesh_addr_t::mip (C++ member), 548
- mesh_ap_cfg_t (C++ class), 552
- mesh_ap_cfg_t::max_connection (C++ member), 552
- mesh_ap_cfg_t::password (C++ member), 552
- MESH_ASSOC_FLAG_NETWORK_FREE (C macro), 555
- MESH_ASSOC_FLAG_ROOT_FIXED (C macro), 555
- MESH_ASSOC_FLAG_ROOTS_FOUND (C macro), 555
- MESH_ASSOC_FLAG_VOTE_IN_PROGRESS (C macro), 555
- mesh_cfg_t (C++ class), 552
- mesh_cfg_t::allow_channel_switch (C++ member), 553
- mesh_cfg_t::channel (C++ member), 553
- mesh_cfg_t::crypto_funcs (C++ member), 553
- mesh_cfg_t::mesh_ap (C++ member), 553
- mesh_cfg_t::mesh_id (C++ member), 553
- mesh_cfg_t::router (C++ member), 553
- MESH_DATA_DROP (C macro), 555
- MESH_DATA_ENC (C macro), 555
- MESH_DATA_FROMDS (C macro), 555
- MESH_DATA_GROUP (C macro), 555
- MESH_DATA_NONBLOCK (C macro), 555
- MESH_DATA_P2P (C macro), 555
- mesh_data_t (C++ class), 552
- mesh_data_t::data (C++ member), 552
- mesh_data_t::proto (C++ member), 552
- mesh_data_t::size (C++ member), 552
- mesh_data_t::tos (C++ member), 552
- MESH_DATA_TODS (C macro), 555
- mesh_disconnect_reason_t (C++ enum), 558
- MESH_EVENT_CHANNEL_SWITCH (C++ enumerator), 556
- mesh_event_channel_switch_t (C++ class), 549
- mesh_event_channel_switch_t::channel (C++ member), 549
- MESH_EVENT_CHILD_CONNECTED (C++ enumerator), 556
- mesh_event_child_connected_t (C++ type), 556
- MESH_EVENT_CHILD_DISCONNECTED (C++ enumerator), 556
- mesh_event_child_disconnected_t (C++ type), 556
- mesh_event_connected_t (C++ class), 549
- mesh_event_connected_t::connected (C++ member), 550
- mesh_event_connected_t::self_layer (C++ member), 550
- mesh_event_disconnected_t (C++ type), 556
- MESH_EVENT_FIND_NETWORK (C++ enumerator), 557
- mesh_event_find_network_t (C++ class), 550
- mesh_event_find_network_t::channel (C++ member), 550
- mesh_event_find_network_t::router_bssid (C++ member), 550
- mesh_event_id_t (C++ enum), 556
- mesh_event_info_t (C++ union), 548
- mesh_event_info_t::channel_switch (C++ member), 548
- mesh_event_info_t::child_connected (C++ member), 548
- mesh_event_info_t::child_disconnected (C++ member), 548
- mesh_event_info_t::connected (C++ member), 548
- mesh_event_info_t::disconnected (C++ member), 548
- mesh_event_info_t::find_network (C++ member), 549
- mesh_event_info_t::layer_change (C++ member), 548
- mesh_event_info_t::network_state (C++ member), 549
- mesh_event_info_t::no_parent (C++ member), 548
- mesh_event_info_t::root_addr (C++ member), 549
- mesh_event_info_t::root_conflict (C++ member), 549
- mesh_event_info_t::root_fixed (C++ member), 549
- mesh_event_info_t::router_switch (C++ member), 549
- mesh_event_info_t::routing_table (C++ member), 548
- mesh_event_info_t::scan_done (C++ member), 549
- mesh_event_info_t::switch_req (C++ member), 549
- mesh_event_info_t::toDS_state (C++ member), 548
- mesh_event_info_t::vote_started (C++ member), 548
- MESH_EVENT_LAYER_CHANGE (C++ enumerator), 556
- mesh_event_layer_change_t (C++ class), 550
- mesh_event_layer_change_t::new_layer (C++ member), 550

- MESH_EVENT_MAX (C++ enumerator), 557
- MESH_EVENT_NETWORK_STATE (C++ enumerator), 557
- mesh_event_network_state_t (C++ class), 551
- mesh_event_network_state_t::is_rootless (C++ member), 551
- MESH_EVENT_NO_PARENT_FOUND (C++ enumerator), 556
- mesh_event_no_parent_found_t (C++ class), 550
- mesh_event_no_parent_found_t::scan_time (C++ member), 550
- MESH_EVENT_PARENT_CONNECTED (C++ enumerator), 556
- MESH_EVENT_PARENT_DISCONNECTED (C++ enumerator), 556
- MESH_EVENT_ROOT_ADDRESS (C++ enumerator), 557
- mesh_event_root_address_t (C++ type), 556
- MESH_EVENT_ROOT_ASKED_YIELD (C++ enumerator), 557
- mesh_event_root_conflict_t (C++ class), 550
- mesh_event_root_conflict_t::addr (C++ member), 551
- mesh_event_root_conflict_t::capacity (C++ member), 551
- mesh_event_root_conflict_t::rssi (C++ member), 551
- MESH_EVENT_ROOT_FIXED (C++ enumerator), 557
- mesh_event_root_fixed_t (C++ class), 551
- mesh_event_root_fixed_t::is_fixed (C++ member), 551
- MESH_EVENT_ROOT_SWITCH_ACK (C++ enumerator), 557
- MESH_EVENT_ROOT_SWITCH_REQ (C++ enumerator), 557
- mesh_event_root_switch_req_t (C++ class), 550
- mesh_event_root_switch_req_t::rc_addr (C++ member), 550
- mesh_event_root_switch_req_t::reason (C++ member), 550
- MESH_EVENT_ROUTER_SWITCH (C++ enumerator), 557
- mesh_event_router_switch_t (C++ type), 556
- MESH_EVENT_ROUTING_TABLE_ADD (C++ enumerator), 556
- mesh_event_routing_table_change_t (C++ class), 551
- mesh_event_routing_table_change_t::rt_mesh_reason (C++ member), 551
- mesh_event_routing_table_change_t::rt_mesh_reason (C++ member), 551
- MESH_EVENT_ROUTING_TABLE_REMOVE (C++ enumerator), 556
- MESH_EVENT_SCAN_DONE (C++ enumerator), 557
- mesh_event_scan_done_t (C++ class), 551
- mesh_event_scan_done_t::number (C++ member), 551
- MESH_EVENT_STARTED (C++ enumerator), 556
- MESH_EVENT_STOP_RECONNECTION (C++ enumerator), 557
- MESH_EVENT_STOPPED (C++ enumerator), 556
- MESH_EVENT_TODS_STATE (C++ enumerator), 556
- mesh_event_toDS_state_t (C++ enum), 559
- MESH_EVENT_VOTE_STARTED (C++ enumerator), 556
- mesh_event_vote_started_t (C++ class), 550
- mesh_event_vote_started_t::attempts (C++ member), 550
- mesh_event_vote_started_t::rc_addr (C++ member), 550
- mesh_event_vote_started_t::reason (C++ member), 550
- MESH_EVENT_VOTE_STOPPED (C++ enumerator), 556
- MESH_IDLE (C++ enumerator), 557
- MESH_INIT_CONFIG_DEFAULT (C macro), 556
- MESH_LEAF (C++ enumerator), 557
- MESH_MPS (C macro), 554
- MESH_MTU (C macro), 554
- MESH_NODE (C++ enumerator), 557
- MESH_OPT_RECV_DS_ADDR (C macro), 555
- MESH_OPT_SEND_GROUP (C macro), 555
- mesh_opt_t (C++ class), 551
- mesh_opt_t::len (C++ member), 551
- mesh_opt_t::type (C++ member), 551
- mesh_opt_t::val (C++ member), 552
- MESH_PROTO_BIN (C++ enumerator), 557
- MESH_PROTO_HTTP (C++ enumerator), 557
- MESH_PROTO_JSON (C++ enumerator), 558
- MESH_PROTO_MQTT (C++ enumerator), 558
- mesh_proto_t (C++ enum), 557
- mesh_rc_config_t (C++ union), 549
- mesh_rc_config_t::attempts (C++ member), 549
- mesh_rc_config_t::rc_addr (C++ member), 549
- MESH_REASON_CYCLIC (C++ enumerator), 558
- MESH_REASON_DIFF_ID (C++ enumerator), 558
- MESH_REASON_EMPTY_PASSWORD (C++ enumerator), 558
- MESH_REASON_IE_UNKNOWN (C++ enumerator), 558
- MESH_REASON_LEAF (C++ enumerator), 558
- MESH_REASON_PARENT_IDLE (C++ enumerator), 558
- MESH_REASON_PARENT_STOPPED (C++ enumerator), 558
- MESH_REASON_PARENT_UNENCRYPTED (C++ enumerator), 559
- MESH_REASON_PARENT_WORSE (C++ enumerator), 558
- MESH_REASON_ROOTS (C++ enumerator), 558

- MESH_REASON_SCAN_FAIL (C++ enumerator), 558
- MESH_REASON_WAIVE_ROOT (C++ enumerator), 558
- MESH_ROOT (C++ enumerator), 557
- MESH_ROOT_LAYER (C macro), 554
- mesh_router_t (C++ class), 552
- mesh_router_t::allow_router_switch (C++ member), 552
- mesh_router_t::bssid (C++ member), 552
- mesh_router_t::password (C++ member), 552
- mesh_router_t::ssid (C++ member), 552
- mesh_router_t::ssid_len (C++ member), 552
- mesh_rx_pending_t (C++ class), 553
- mesh_rx_pending_t::toDS (C++ member), 554
- mesh_rx_pending_t::toSelf (C++ member), 554
- MESH_STA (C++ enumerator), 557
- MESH_TODS_REACHABLE (C++ enumerator), 559
- MESH_TODS_UNREACHABLE (C++ enumerator), 559
- MESH_TOS_DEF (C++ enumerator), 558
- MESH_TOS_E2E (C++ enumerator), 558
- MESH_TOS_P2P (C++ enumerator), 558
- mesh_tos_t (C++ enum), 558
- mesh_tx_pending_t (C++ class), 553
- mesh_tx_pending_t::broadcast (C++ member), 553
- mesh_tx_pending_t::mgmt (C++ member), 553
- mesh_tx_pending_t::to_child (C++ member), 553
- mesh_tx_pending_t::to_child_p2p (C++ member), 553
- mesh_tx_pending_t::to_parent (C++ member), 553
- mesh_tx_pending_t::to_parent_p2p (C++ member), 553
- mesh_type_t (C++ enum), 557
- MESH_VOTE_REASON_CHILD_INITIATED (C++ enumerator), 558
- MESH_VOTE_REASON_ROOT_INITIATED (C++ enumerator), 558
- mesh_vote_reason_t (C++ enum), 558
- mesh_vote_t (C++ class), 553
- mesh_vote_t::config (C++ member), 553
- mesh_vote_t::is_rc_specified (C++ member), 553
- mesh_vote_t::percentage (C++ member), 553
- mip_t (C++ class), 549
- mip_t::ip4 (C++ member), 549
- mip_t::port (C++ member), 549
- MQTT_CONNECTION_ACCEPTED (C++ enumerator), 810
- MQTT_CONNECTION_REFUSE_BAD_USERNAME (C++ enumerator), 810
- MQTT_CONNECTION_REFUSE_ID_REJECTED (C++ enumerator), 810
- MQTT_CONNECTION_REFUSE_NOT_AUTHORIZED (C++ enumerator), 810
- MQTT_CONNECTION_REFUSE_PROTOCOL (C++ enumerator), 810
- MQTT_CONNECTION_REFUSE_SERVER_UNAVAILABLE (C++ enumerator), 810
- MQTT_ERROR_TYPE_CONNECTION_REFUSED (C++ enumerator), 810
- MQTT_ERROR_TYPE_ESP_TLS (C macro), 809
- MQTT_ERROR_TYPE_NONE (C++ enumerator), 810
- MQTT_ERROR_TYPE_TCP_TRANSPORT (C++ enumerator), 810
- MQTT_EVENT_ANY (C++ enumerator), 809
- MQTT_EVENT_BEFORE_CONNECT (C++ enumerator), 810
- mqtt_event_callback_t (C++ type), 809
- MQTT_EVENT_CONNECTED (C++ enumerator), 809
- MQTT_EVENT_DATA (C++ enumerator), 809
- MQTT_EVENT_DELETED (C++ enumerator), 810
- MQTT_EVENT_DISCONNECTED (C++ enumerator), 809
- MQTT_EVENT_ERROR (C++ enumerator), 809
- MQTT_EVENT_PUBLISHED (C++ enumerator), 809
- MQTT_EVENT_SUBSCRIBED (C++ enumerator), 809
- MQTT_EVENT_UNSUBSCRIBED (C++ enumerator), 809
- MQTT_PROTOCOL_UNDEFINED (C++ enumerator), 811
- MQTT_PROTOCOL_V_3_1 (C++ enumerator), 811
- MQTT_PROTOCOL_V_3_1_1 (C++ enumerator), 811
- MQTT_TRANSPORT_OVER_SSL (C++ enumerator), 810
- MQTT_TRANSPORT_OVER_TCP (C++ enumerator), 810
- MQTT_TRANSPORT_OVER_WS (C++ enumerator), 810
- MQTT_TRANSPORT_OVER_WSS (C++ enumerator), 810
- MQTT_TRANSPORT_UNKNOWN (C++ enumerator), 810
- multi_heap_aligned_alloc (C++ function), 1140
- multi_heap_aligned_free (C++ function), 1140
- multi_heap_check (C++ function), 1141
- multi_heap_dump (C++ function), 1141
- multi_heap_free (C++ function), 1140
- multi_heap_free_size (C++ function), 1141
- multi_heap_get_allocated_size (C++ function), 1140
- multi_heap_get_info (C++ function), 1142
- multi_heap_handle_t (C++ type), 1142
- multi_heap_info_t (C++ class), 1142
- multi_heap_info_t::allocated_blocks (C++ member), 1142
- multi_heap_info_t::free_blocks (C++ member), 1142
- multi_heap_info_t::largest_free_block (C++ member), 1142
- multi_heap_info_t::minimum_free_bytes (C++ member), 1142

- multi_heap_info_t::total_allocated_bytes (C++ member), 1142
 multi_heap_info_t::total_blocks (C++ member), 1142
 multi_heap_info_t::total_free_bytes (C++ member), 1142
 multi_heap_malloc (C++ function), 1140
 multi_heap_minimum_free_size (C++ function), 1142
 multi_heap_realloc (C++ function), 1140
 multi_heap_register (C++ function), 1141
 multi_heap_set_lock (C++ function), 1141
- ## N
- name_uuid (C++ class), 899
 name_uuid::name (C++ member), 899
 name_uuid::uuid (C++ member), 899
 nvs_close (C++ function), 942
 nvs_commit (C++ function), 941
 NVS_DEFAULT_PART_NAME (C macro), 945
 nvs_entry_find (C++ function), 943
 nvs_entry_info (C++ function), 943
 nvs_entry_info_t (C++ class), 944
 nvs_entry_info_t::key (C++ member), 944
 nvs_entry_info_t::namespace_name (C++ member), 944
 nvs_entry_info_t::type (C++ member), 944
 nvs_entry_next (C++ function), 943
 nvs_erase_all (C++ function), 941
 nvs_erase_key (C++ function), 941
 nvs_flash_deinit (C++ function), 935
 nvs_flash_deinit_partition (C++ function), 935
 nvs_flash_erase (C++ function), 935
 nvs_flash_erase_partition (C++ function), 935
 nvs_flash_erase_partition_ptr (C++ function), 935
 nvs_flash_generate_keys (C++ function), 936
 nvs_flash_init (C++ function), 934
 nvs_flash_init_partition (C++ function), 934
 nvs_flash_init_partition_ptr (C++ function), 934
 nvs_flash_read_security_cfg (C++ function), 936
 nvs_flash_secure_init (C++ function), 936
 nvs_flash_secure_init_partition (C++ function), 936
 nvs_get_blob (C++ function), 940
 nvs_get_i16 (C++ function), 939
 nvs_get_i32 (C++ function), 939
 nvs_get_i64 (C++ function), 939
 nvs_get_i8 (C++ function), 938
 nvs_get_stats (C++ function), 942
 nvs_get_str (C++ function), 939
 nvs_get_u16 (C++ function), 939
 nvs_get_u32 (C++ function), 939
 nvs_get_u64 (C++ function), 939
 nvs_get_u8 (C++ function), 939
 nvs_get_used_entry_count (C++ function), 942
 nvs_handle (C++ type), 945
 nvs_handle_t (C++ type), 945
 nvs_iterator_t (C++ type), 945
 NVS_KEY_SIZE (C macro), 937
 nvs_open (C++ function), 940
 nvs_open_from_partition (C++ function), 940
 nvs_open_mode (C++ type), 945
 nvs_open_mode_t (C++ enum), 946
 NVS_PART_NAME_MAX_SIZE (C macro), 945
 NVS_READONLY (C++ enumerator), 946
 NVS_READWRITE (C++ enumerator), 946
 nvs_release_iterator (C++ function), 944
 nvs_sec_cfg_t (C++ class), 936
 nvs_sec_cfg_t::key (C++ member), 937
 nvs_sec_cfg_t::tky (C++ member), 937
 nvs_set_blob (C++ function), 941
 nvs_set_i16 (C++ function), 937
 nvs_set_i32 (C++ function), 937
 nvs_set_i64 (C++ function), 937
 nvs_set_i8 (C++ function), 937
 nvs_set_str (C++ function), 938
 nvs_set_u16 (C++ function), 937
 nvs_set_u32 (C++ function), 937
 nvs_set_u64 (C++ function), 938
 nvs_set_u8 (C++ function), 937
 nvs_stats_t (C++ class), 944
 nvs_stats_t::free_entries (C++ member), 944
 nvs_stats_t::namespace_count (C++ member), 944
 nvs_stats_t::total_entries (C++ member), 944
 nvs_stats_t::used_entries (C++ member), 944
 NVS_TYPE_ANY (C++ enumerator), 946
 NVS_TYPE_BLOB (C++ enumerator), 946
 NVS_TYPE_I16 (C++ enumerator), 946
 NVS_TYPE_I32 (C++ enumerator), 946
 NVS_TYPE_I64 (C++ enumerator), 946
 NVS_TYPE_I8 (C++ enumerator), 946
 NVS_TYPE_STR (C++ enumerator), 946
 nvs_type_t (C++ enum), 946
 NVS_TYPE_U16 (C++ enumerator), 946
 NVS_TYPE_U32 (C++ enumerator), 946
 NVS_TYPE_U64 (C++ enumerator), 946
 NVS_TYPE_U8 (C++ enumerator), 946
- ## O
- OTA_SIZE_UNKNOWN (C macro), 1187
- ## P
- PCNT_CHANNEL_0 (C++ enumerator), 695
 PCNT_CHANNEL_1 (C++ enumerator), 695
 PCNT_CHANNEL_MAX (C++ enumerator), 695

- pcnt_channel_t (C++ enum), 695
 pcnt_config_t (C++ class), 693
 pcnt_config_t::channel (C++ member), 694
 pcnt_config_t::counter_h_lim (C++ member), 694
 pcnt_config_t::counter_l_lim (C++ member), 694
 pcnt_config_t::ctrl_gpio_num (C++ member), 694
 pcnt_config_t::hctrl_mode (C++ member), 694
 pcnt_config_t::lctrl_mode (C++ member), 694
 pcnt_config_t::neg_mode (C++ member), 694
 pcnt_config_t::pos_mode (C++ member), 694
 pcnt_config_t::pulse_gpio_num (C++ member), 693
 pcnt_config_t::unit (C++ member), 694
 PCNT_COUNT_DEC (C++ enumerator), 694
 PCNT_COUNT_DIS (C++ enumerator), 694
 PCNT_COUNT_INC (C++ enumerator), 694
 PCNT_COUNT_MAX (C++ enumerator), 695
 pcnt_count_mode_t (C++ enum), 694
 pcnt_counter_clear (C++ function), 690
 pcnt_counter_pause (C++ function), 689
 pcnt_counter_resume (C++ function), 689
 pcnt_ctrl_mode_t (C++ enum), 694
 pcnt_event_disable (C++ function), 690
 pcnt_event_enable (C++ function), 690
 PCNT_EVT_H_LIM (C++ enumerator), 695
 PCNT_EVT_L_LIM (C++ enumerator), 695
 PCNT_EVT_MAX (C++ enumerator), 695
 PCNT_EVT_THRES_0 (C++ enumerator), 695
 PCNT_EVT_THRES_1 (C++ enumerator), 695
 pcnt_evt_type_t (C++ enum), 695
 PCNT_EVT_ZERO (C++ enumerator), 695
 pcnt_filter_disable (C++ function), 692
 pcnt_filter_enable (C++ function), 691
 pcnt_get_counter_value (C++ function), 689
 pcnt_get_event_value (C++ function), 691
 pcnt_get_filter_value (C++ function), 692
 pcnt_intr_disable (C++ function), 690
 pcnt_intr_enable (C++ function), 690
 pcnt_isr_handle_t (C++ type), 693
 pcnt_isr_handler_add (C++ function), 692
 pcnt_isr_handler_remove (C++ function), 693
 pcnt_isr_register (C++ function), 691
 pcnt_isr_service_install (C++ function), 693
 pcnt_isr_service_uninstall (C++ function), 693
 PCNT_MODE_DISABLE (C++ enumerator), 694
 PCNT_MODE_KEEP (C++ enumerator), 694
 PCNT_MODE_MAX (C++ enumerator), 694
 PCNT_MODE_REVERSE (C++ enumerator), 694
 pcnt_port_t (C++ type), 694
 pcnt_set_event_value (C++ function), 691
 pcnt_set_filter_value (C++ function), 692
 pcnt_set_mode (C++ function), 692
 pcnt_set_pin (C++ function), 691
 pcnt_unit_config (C++ function), 689
 pcnt_unit_t (C++ type), 694
 pcQueueGetName (C++ function), 1065
 pcTaskGetTaskName (C++ function), 1047
 pcTimerGetTimerName (C++ function), 1098
 PDM_PCM_CONV_DISABLE (C++ enumerator), 655
 PDM_PCM_CONV_ENABLE (C++ enumerator), 655
 pdm_pcm_conv_t (C++ enum), 655
 PDM_SAMPLE_RATE_RATIO_128 (C++ enumerator), 655
 PDM_SAMPLE_RATE_RATIO_64 (C++ enumerator), 655
 pdm_sample_rate_ratio_t (C++ enum), 655
 PendedFunction_t (C++ type), 1107
 protocomm_add_endpoint (C++ function), 893
 protocomm_ble_config (C++ class), 899
 protocomm_ble_config::device_name (C++ member), 899
 protocomm_ble_config::nu_lookup (C++ member), 899
 protocomm_ble_config::nu_lookup_count (C++ member), 899
 protocomm_ble_config::service_uuid (C++ member), 899
 protocomm_ble_config_t (C++ type), 899
 protocomm_ble_name_uuid_t (C++ type), 899
 protocomm_ble_start (C++ function), 898
 protocomm_ble_stop (C++ function), 898
 protocomm_close_session (C++ function), 894
 protocomm_delete (C++ function), 893
 protocomm_http_server_config_t (C++ class), 898
 protocomm_http_server_config_t::port (C++ member), 898
 protocomm_http_server_config_t::stack_size (C++ member), 898
 protocomm_http_server_config_t::task_priority (C++ member), 898
 protocomm_httpd_config_data_t (C++ union), 897
 protocomm_httpd_config_data_t::config (C++ member), 897
 protocomm_httpd_config_data_t::handle (C++ member), 897
 protocomm_httpd_config_t (C++ class), 898
 protocomm_httpd_config_t::data (C++ member), 898
 protocomm_httpd_config_t::ext_handle_provided (C++ member), 898
 PROTOCOL_HTTPD_DEFAULT_CONFIG (C macro), 898
 protocomm_httpd_start (C++ function), 897
 protocomm_httpd_stop (C++ function), 897
 protocomm_new (C++ function), 893
 protocomm_open_session (C++ function), 893
 protocomm_remove_endpoint (C++ function),

893

protocomm_req_handle (C++ function), 894

protocomm_req_handler_t (C++ type), 895

protocomm_security (C++ class), 896

protocomm_security::cleanup (C++ member), 896

protocomm_security::close_transport_session (C++ member), 896

protocomm_security::decrypt (C++ member), 896

protocomm_security::encrypt (C++ member), 896

protocomm_security::init (C++ member), 896

protocomm_security::new_transport_session (C++ member), 896

protocomm_security::security_req_handler (C++ member), 896

protocomm_security::ver (C++ member), 896

protocomm_security_handle_t (C++ type), 896

protocomm_security_pop (C++ class), 896

protocomm_security_pop::data (C++ member), 896

protocomm_security_pop::len (C++ member), 896

protocomm_security_pop_t (C++ type), 896

protocomm_security_t (C++ type), 896

protocomm_set_security (C++ function), 894

protocomm_set_version (C++ function), 895

protocomm_t (C++ type), 895

protocomm_unset_security (C++ function), 895

protocomm_unset_version (C++ function), 895

PROV_DATA_FLAGS_FLAG (C macro), 311

PROV_DATA_IV_INDEX_FLAG (C macro), 311

PROV_DATA_NET_IDX_FLAG (C macro), 311

PROXY_FILTER_BLACKLIST (C++ enumerator), 325

PROXY_FILTER_WHITELIST (C++ enumerator), 325

psk_hint_key_t (C++ type), 819

psk_key_hint (C++ class), 816

psk_key_hint::hint (C++ member), 816

psk_key_hint::key (C++ member), 816

psk_key_hint::key_size (C++ member), 816

PTHREAD_STACK_MIN (C macro), 1024

pvTaskGetThreadLocalStoragePointer (C++ function), 1048

pvTimerGetTimerID (C++ function), 1095

pxTaskGetStackStart (C++ function), 1047

Q

QueueHandle_t (C++ type), 1078

QueueSetHandle_t (C++ type), 1078

QueueSetMemberHandle_t (C++ type), 1078

R

R0 (C macro), 1670

R1 (C macro), 1670

R2 (C macro), 1670

R3 (C macro), 1670

RINGBUF_TYPE_ALLOWSPLIT (C++ enumerator), 1129

RINGBUF_TYPE_BYTEBUF (C++ enumerator), 1129

RINGBUF_TYPE_MAX (C++ enumerator), 1129

RINGBUF_TYPE_NOSPLIT (C++ enumerator), 1129

RingbufferType_t (C++ enum), 1129

RingbufHandle_t (C++ type), 1129

RMT_BASECLK_APB (C++ enumerator), 710

RMT_BASECLK_MAX (C++ enumerator), 710

RMT_BASECLK_REF (C++ enumerator), 710

RMT_CARRIER_LEVEL_HIGH (C++ enumerator), 711

RMT_CARRIER_LEVEL_LOW (C++ enumerator), 711

RMT_CARRIER_LEVEL_MAX (C++ enumerator), 711

rmt_carrier_level_t (C++ enum), 711

RMT_CHANNEL_BUSY (C++ enumerator), 711

RMT_CHANNEL_IDLE (C++ enumerator), 711

rmt_channel_status_result_t (C++ class), 709

rmt_channel_status_result_t::status (C++ member), 710

rmt_channel_status_t (C++ enum), 711

rmt_channel_t (C++ type), 710

RMT_CHANNEL_UNINIT (C++ enumerator), 711

rmt_clr_intr_enable_mask (C++ function), 704

rmt_config (C++ function), 705

rmt_config_t (C++ class), 708

rmt_config_t::channel (C++ member), 708

rmt_config_t::clk_div (C++ member), 708

rmt_config_t::gpio_num (C++ member), 708

rmt_config_t::mem_block_num (C++ member), 708

rmt_config_t::rmt_mode (C++ member), 708

rmt_config_t::rx_config (C++ member), 709

rmt_config_t::tx_config (C++ member), 708

RMT_DATA_MODE_FIFO (C++ enumerator), 710

RMT_DATA_MODE_MAX (C++ enumerator), 710

RMT_DATA_MODE_MEM (C++ enumerator), 710

rmt_data_mode_t (C++ enum), 710

RMT_DEFAULT_CONFIG_RX (C macro), 709

RMT_DEFAULT_CONFIG_TX (C macro), 709

rmt_driver_install (C++ function), 705

rmt_driver_uninstall (C++ function), 706

rmt_fill_tx_items (C++ function), 705

rmt_get_channel_status (C++ function), 706

rmt_get_clk_div (C++ function), 699

rmt_get_counter_clock (C++ function), 706

rmt_get_idle_level (C++ function), 703

rmt_get_mem_block_num (C++ function), 700

rmt_get_mem_pd (C++ function), 701

rmt_get_memory_owner (C++ function), 702

rmt_get_ringbuf_handle (C++ function), 707

- `rmt_get_rx_idle_thresh` (C++ function), 700
`rmt_get_source_clk` (C++ function), 703
`rmt_get_status` (C++ function), 704
`rmt_get_tx_loop_mode` (C++ function), 702
`RMT_IDLE_LEVEL_HIGH` (C++ enumerator), 711
`RMT_IDLE_LEVEL_LOW` (C++ enumerator), 710
`RMT_IDLE_LEVEL_MAX` (C++ enumerator), 711
`rmt_idle_level_t` (C++ enum), 710
`rmt_isr_deregister` (C++ function), 705
`rmt_isr_handle_t` (C++ type), 709
`rmt_isr_register` (C++ function), 705
`RMT_MEM_ITEM_NUM` (C macro), 709
`RMT_MEM_OWNER_MAX` (C++ enumerator), 710
`RMT_MEM_OWNER_RX` (C++ enumerator), 710
`rmt_mem_owner_t` (C++ enum), 710
`RMT_MEM_OWNER_TX` (C++ enumerator), 710
`rmt_memory_rw_rst` (C++ function), 702
`RMT_MODE_MAX` (C++ enumerator), 710
`RMT_MODE_RX` (C++ enumerator), 710
`rmt_mode_t` (C++ enum), 710
`RMT_MODE_TX` (C++ enumerator), 710
`rmt_register_tx_end_callback` (C++ function), 707
`rmt_rx_config_t` (C++ class), 708
`rmt_rx_config_t::filter_en` (C++ member), 708
`rmt_rx_config_t::filter_ticks_thresh` (C++ member), 708
`rmt_rx_config_t::idle_threshold` (C++ member), 708
`rmt_rx_start` (C++ function), 701
`rmt_rx_stop` (C++ function), 702
`rmt_set_clk_div` (C++ function), 699
`rmt_set_err_intr_en` (C++ function), 704
`rmt_set_idle_level` (C++ function), 703
`rmt_set_intr_enable_mask` (C++ function), 704
`rmt_set_mem_block_num` (C++ function), 700
`rmt_set_mem_pd` (C++ function), 701
`rmt_set_memory_owner` (C++ function), 702
`rmt_set_pin` (C++ function), 705
`rmt_set_rx_filter` (C++ function), 703
`rmt_set_rx_idle_thresh` (C++ function), 700
`rmt_set_rx_intr_en` (C++ function), 704
`rmt_set_source_clk` (C++ function), 703
`rmt_set_tx_carrier` (C++ function), 700
`rmt_set_tx_intr_en` (C++ function), 704
`rmt_set_tx_loop_mode` (C++ function), 702
`rmt_set_tx_thr_intr_en` (C++ function), 704
`rmt_source_clk_t` (C++ enum), 710
`rmt_translator_init` (C++ function), 707
`rmt_tx_config_t` (C++ class), 708
`rmt_tx_config_t::carrier_duty_percent` (C++ member), 708
`rmt_tx_config_t::carrier_en` (C++ member), 708
`rmt_tx_config_t::carrier_freq_hz` (C++ member), 708
`rmt_tx_config_t::carrier_level` (C++ member), 708
`rmt_tx_config_t::idle_level` (C++ member), 708
`rmt_tx_config_t::idle_output_en` (C++ member), 708
`rmt_tx_config_t::loop_en` (C++ member), 708
`rmt_tx_end_callback_t` (C++ class), 709
`rmt_tx_end_callback_t::arg` (C++ member), 709
`rmt_tx_end_callback_t::function` (C++ member), 709
`rmt_tx_end_fn_t` (C++ type), 709
`rmt_tx_start` (C++ function), 701
`rmt_tx_stop` (C++ function), 701
`rmt_wait_tx_done` (C++ function), 707
`rmt_write_items` (C++ function), 706
`rmt_write_sample` (C++ function), 707
`ROLE_FAST_PROV` (C++ enumerator), 325
`ROLE_NODE` (C++ enumerator), 325
`ROLE_PROVISIONER` (C++ enumerator), 325
`rtc_gpio_deinit` (C++ function), 630
`rtc_gpio_force_hold_all` (C++ function), 632
`rtc_gpio_force_hold_dis_all` (C++ function), 632
`rtc_gpio_get_drive_capability` (C++ function), 632
`rtc_gpio_get_level` (C++ function), 630
`rtc_gpio_hold_dis` (C++ function), 632
`rtc_gpio_hold_en` (C++ function), 631
`rtc_gpio_init` (C++ function), 630
`RTC_GPIO_IS_VALID_GPIO` (C macro), 633
`rtc_gpio_is_valid_gpio` (C++ function), 630
`rtc_gpio_isolate` (C++ function), 632
`RTC_GPIO_MODE_DISABLED` (C++ enumerator), 633
`RTC_GPIO_MODE_INPUT_ONLY` (C++ enumerator), 633
`RTC_GPIO_MODE_INPUT_OUTPUT` (C++ enumerator), 633
`RTC_GPIO_MODE_INPUT_OUTPUT_OD` (C++ enumerator), 633
`RTC_GPIO_MODE_OUTPUT_OD` (C++ enumerator), 633
`RTC_GPIO_MODE_OUTPUT_ONLY` (C++ enumerator), 633
`rtc_gpio_mode_t` (C++ enum), 633
`rtc_gpio_pullup_dis` (C++ function), 631
`rtc_gpio_pullup_en` (C++ function), 631
`rtc_gpio_set_direction` (C++ function), 630
`rtc_gpio_set_direction_in_sleep` (C++ function), 630
`rtc_gpio_set_drive_capability` (C++ function), 632
`rtc_gpio_set_level` (C++ function), 630

- [rtc_gpio_wakeup_disable \(C++ function\), 633](#)
[rtc_gpio_wakeup_enable \(C++ function\), 632](#)
[rtc_io_number_get \(C++ function\), 630](#)
[RTC_SLOW_MEM \(C macro\), 1673](#)
- ## S
- [sample_to_rmt_t \(C++ type\), 709](#)
[SC_EVENT_FOUND_CHANNEL \(C++ enumerator\), 524](#)
[SC_EVENT_GOT_SSID_PSWD \(C++ enumerator\), 524](#)
[SC_EVENT_SCAN_DONE \(C++ enumerator\), 524](#)
[SC_EVENT_SEND_ACK_DONE \(C++ enumerator\), 524](#)
[SC_TYPE_AIRKISS \(C++ enumerator\), 524](#)
[SC_TYPE_ESPTOUCH \(C++ enumerator\), 524](#)
[SC_TYPE_ESPTOUCH_AIRKISS \(C++ enumerator\), 524](#)
[sdio_event_cb_t \(C++ type\), 732](#)
[sdio_slave_buf_handle_t \(C++ type\), 732](#)
[sdio_slave_clear_host_int \(C++ function\), 731](#)
[sdio_slave_config_t \(C++ class\), 731](#)
[sdio_slave_config_t::event_cb \(C++ member\), 732](#)
[sdio_slave_config_t::flags \(C++ member\), 732](#)
[sdio_slave_config_t::recv_buffer_size \(C++ member\), 732](#)
[sdio_slave_config_t::send_queue_size \(C++ member\), 731](#)
[sdio_slave_config_t::sending_mode \(C++ member\), 731](#)
[sdio_slave_config_t::timing \(C++ member\), 731](#)
[sdio_slave_deinit \(C++ function\), 729](#)
[SDIO_SLAVE_FLAG_DAT2_DISABLED \(C macro\), 732](#)
[SDIO_SLAVE_FLAG_HOST_INTR_DISABLED \(C macro\), 732](#)
[SDIO_SLAVE_FLAG_INTERNAL_PULLUP \(C macro\), 732](#)
[sdio_slave_get_host_intena \(C++ function\), 731](#)
[SDIO_SLAVE_HOSTINT_BIT0 \(C++ enumerator\), 728](#)
[SDIO_SLAVE_HOSTINT_BIT1 \(C++ enumerator\), 728](#)
[SDIO_SLAVE_HOSTINT_BIT2 \(C++ enumerator\), 728](#)
[SDIO_SLAVE_HOSTINT_BIT3 \(C++ enumerator\), 728](#)
[SDIO_SLAVE_HOSTINT_BIT4 \(C++ enumerator\), 728](#)
[SDIO_SLAVE_HOSTINT_BIT5 \(C++ enumerator\), 728](#)
[SDIO_SLAVE_HOSTINT_BIT6 \(C++ enumerator\), 728](#)
[SDIO_SLAVE_HOSTINT_BIT7 \(C++ enumerator\), 728](#)
[SDIO_SLAVE_HOSTINT_SEND_NEW_PACKET \(C++ enumerator\), 728](#)
[sdio_slave_hostint_t \(C++ enum\), 728](#)
[sdio_slave_initialize \(C++ function\), 729](#)
[sdio_slave_read_reg \(C++ function\), 730](#)
[sdio_slave_recv \(C++ function\), 730](#)
[sdio_slave_recv_get_buf \(C++ function\), 730](#)
[sdio_slave_recv_load_buf \(C++ function\), 729](#)
[SDIO_SLAVE_RECV_MAX_BUFFER \(C macro\), 732](#)
[sdio_slave_recv_register_buf \(C++ function\), 729](#)
[sdio_slave_recv_unregister_buf \(C++ function\), 729](#)
[sdio_slave_reset \(C++ function\), 729](#)
[sdio_slave_send_get_finished \(C++ function\), 730](#)
[sdio_slave_send_host_int \(C++ function\), 731](#)
[SDIO_SLAVE_SEND_PACKET \(C++ enumerator\), 728](#)
[sdio_slave_send_queue \(C++ function\), 730](#)
[SDIO_SLAVE_SEND_STREAM \(C++ enumerator\), 728](#)
[sdio_slave_sending_mode_t \(C++ enum\), 728](#)
[sdio_slave_set_host_intena \(C++ function\), 731](#)
[sdio_slave_start \(C++ function\), 729](#)
[sdio_slave_stop \(C++ function\), 729](#)
[SDIO_SLAVE_TIMING_NSEND_NSAMPLE \(C++ enumerator\), 728](#)
[SDIO_SLAVE_TIMING_NSEND_PSAMPLE \(C++ enumerator\), 728](#)
[SDIO_SLAVE_TIMING_PSEND_NSAMPLE \(C++ enumerator\), 728](#)
[SDIO_SLAVE_TIMING_PSEND_PSAMPLE \(C++ enumerator\), 728](#)
[sdio_slave_timing_t \(C++ enum\), 728](#)
[sdio_slave_transmit \(C++ function\), 730](#)
[sdio_slave_wait_int \(C++ function\), 731](#)
[sdio_slave_write_reg \(C++ function\), 731](#)
[sdmmc_card_init \(C++ function\), 955](#)
[sdmmc_card_print_info \(C++ function\), 955](#)
[sdmmc_card_t \(C++ class\), 961](#)
[sdmmc_card_t::cid \(C++ member\), 961](#)
[sdmmc_card_t::csd \(C++ member\), 961](#)
[sdmmc_card_t::ext_csd \(C++ member\), 961](#)
[sdmmc_card_t::host \(C++ member\), 961](#)
[sdmmc_card_t::is_ddr \(C++ member\), 961](#)
[sdmmc_card_t::is_mem \(C++ member\), 961](#)
[sdmmc_card_t::is_mmc \(C++ member\), 961](#)
[sdmmc_card_t::is_sdio \(C++ member\), 961](#)
[sdmmc_card_t::log_bus_width \(C++ member\), 961](#)
[sdmmc_card_t::max_freq_khz \(C++ member\), 961](#)

- sdmmc_card_t::num_io_functions (C++ member), 961
- sdmmc_card_t::ocr (C++ member), 961
- sdmmc_card_t::raw_cid (C++ member), 961
- sdmmc_card_t::rca (C++ member), 961
- sdmmc_card_t::reserved (C++ member), 961
- sdmmc_card_t::scr (C++ member), 961
- sdmmc_cid_t (C++ class), 959
- sdmmc_cid_t::date (C++ member), 959
- sdmmc_cid_t::mfg_id (C++ member), 959
- sdmmc_cid_t::name (C++ member), 959
- sdmmc_cid_t::oem_id (C++ member), 959
- sdmmc_cid_t::revision (C++ member), 959
- sdmmc_cid_t::serial (C++ member), 959
- sdmmc_command_t (C++ class), 959
- sdmmc_command_t::arg (C++ member), 960
- sdmmc_command_t::blklen (C++ member), 960
- sdmmc_command_t::data (C++ member), 960
- sdmmc_command_t::datalen (C++ member), 960
- sdmmc_command_t::error (C++ member), 960
- sdmmc_command_t::flags (C++ member), 960
- sdmmc_command_t::opcode (C++ member), 960
- sdmmc_command_t::response (C++ member), 960
- sdmmc_command_t::timeout_ms (C++ member), 960
- sdmmc_csd_t (C++ class), 958
- sdmmc_csd_t::capacity (C++ member), 958
- sdmmc_csd_t::card_command_class (C++ member), 959
- sdmmc_csd_t::csd_ver (C++ member), 958
- sdmmc_csd_t::mmc_ver (C++ member), 958
- sdmmc_csd_t::read_block_len (C++ member), 959
- sdmmc_csd_t::sector_size (C++ member), 958
- sdmmc_csd_t::tr_speed (C++ member), 959
- sdmmc_ext_csd_t (C++ class), 959
- sdmmc_ext_csd_t::power_class (C++ member), 959
- SDMMC_FREQ_26M (C macro), 962
- SDMMC_FREQ_52M (C macro), 962
- SDMMC_FREQ_DEFAULT (C macro), 962
- SDMMC_FREQ_HIGHSPEED (C macro), 962
- SDMMC_FREQ_PROBING (C macro), 962
- SDMMC_HOST_DEFAULT (C macro), 719
- sdmmc_host_deinit (C++ function), 718
- sdmmc_host_do_transaction (C++ function), 717
- SDMMC_HOST_FLAG_1BIT (C macro), 962
- SDMMC_HOST_FLAG_4BIT (C macro), 962
- SDMMC_HOST_FLAG_8BIT (C macro), 962
- SDMMC_HOST_FLAG_DDR (C macro), 962
- SDMMC_HOST_FLAG_SPI (C macro), 962
- sdmmc_host_get_slot_width (C++ function), 717
- sdmmc_host_init (C++ function), 716
- sdmmc_host_init_slot (C++ function), 716
- sdmmc_host_io_int_enable (C++ function), 718
- sdmmc_host_io_int_wait (C++ function), 718
- sdmmc_host_pullup_en (C++ function), 718
- sdmmc_host_set_bus_ddr_mode (C++ function), 717
- sdmmc_host_set_bus_width (C++ function), 717
- sdmmc_host_set_card_clk (C++ function), 717
- SDMMC_HOST_SLOT_0 (C macro), 719
- SDMMC_HOST_SLOT_1 (C macro), 719
- sdmmc_host_t (C++ class), 960
- sdmmc_host_t::command_timeout_ms (C++ member), 961
- sdmmc_host_t::deinit (C++ member), 961
- sdmmc_host_t::do_transaction (C++ member), 961
- sdmmc_host_t::flags (C++ member), 960
- sdmmc_host_t::get_bus_width (C++ member), 960
- sdmmc_host_t::init (C++ member), 960
- sdmmc_host_t::io_int_enable (C++ member), 961
- sdmmc_host_t::io_int_wait (C++ member), 961
- sdmmc_host_t::io_voltage (C++ member), 960
- sdmmc_host_t::max_freq_khz (C++ member), 960
- sdmmc_host_t::set_bus_ddr_mode (C++ member), 960
- sdmmc_host_t::set_bus_width (C++ member), 960
- sdmmc_host_t::set_card_clk (C++ member), 960
- sdmmc_host_t::slot (C++ member), 960
- sdmmc_io_enable_int (C++ function), 957
- sdmmc_io_get_cis_data (C++ function), 957
- sdmmc_io_print_cis_info (C++ function), 958
- sdmmc_io_read_blocks (C++ function), 957
- sdmmc_io_read_byte (C++ function), 956
- sdmmc_io_read_bytes (C++ function), 956
- sdmmc_io_wait_int (C++ function), 957
- sdmmc_io_write_blocks (C++ function), 957
- sdmmc_io_write_byte (C++ function), 956
- sdmmc_io_write_bytes (C++ function), 956
- sdmmc_read_sectors (C++ function), 955
- sdmmc_response_t (C++ type), 962
- sdmmc_scr_t (C++ class), 959
- sdmmc_scr_t::bus_width (C++ member), 959
- sdmmc_scr_t::sd_spec (C++ member), 959
- SDMMC_SLOT_CONFIG_DEFAULT (C macro), 719
- sdmmc_slot_config_t (C++ class), 718
- sdmmc_slot_config_t::flags (C++ member), 718
- sdmmc_slot_config_t::gpio_cd (C++ member), 718

- sdmmc_slot_config_t::gpio_wp (C++ member), 718
- sdmmc_slot_config_t::width (C++ member), 718
- SDMMC_SLOT_FLAG_INTERNAL_PULLUP (C macro), 719
- SDMMC_SLOT_NO_CD (C macro), 719
- SDMMC_SLOT_NO_WP (C macro), 719
- SDMMC_SLOT_WIDTH_DEFAULT (C macro), 719
- sdmmc_switch_func_rsp_t (C++ class), 959
- sdmmc_switch_func_rsp_t::data (C++ member), 959
- sdmmc_write_sectors (C++ function), 955
- SDSPI_HOST_DEFAULT (C macro), 722
- sdspi_host_deinit (C++ function), 721
- sdspi_host_do_transaction (C++ function), 720
- sdspi_host_init (C++ function), 720
- sdspi_host_init_slot (C++ function), 720
- sdspi_host_io_int_enable (C++ function), 721
- sdspi_host_io_int_wait (C++ function), 721
- sdspi_host_set_card_clk (C++ function), 720
- SDSPI_SLOT_CONFIG_DEFAULT (C macro), 722
- sdspi_slot_config_t (C++ class), 721
- sdspi_slot_config_t::dma_channel (C++ member), 721
- sdspi_slot_config_t::gpio_cd (C++ member), 721
- sdspi_slot_config_t::gpio_cs (C++ member), 721
- sdspi_slot_config_t::gpio_int (C++ member), 721
- sdspi_slot_config_t::gpio_miso (C++ member), 721
- sdspi_slot_config_t::gpio_mosi (C++ member), 721
- sdspi_slot_config_t::gpio_sck (C++ member), 721
- sdspi_slot_config_t::gpio_wp (C++ member), 721
- SDSPI_SLOT_NO_CD (C macro), 722
- SDSPI_SLOT_NO_INT (C macro), 722
- SDSPI_SLOT_NO_WP (C macro), 722
- SemaphoreHandle_t (C++ type), 1091
- semBINARY_SEMAPHORE_QUEUE_LENGTH (C macro), 1079
- semGIVE_BLOCK_TIME (C macro), 1079
- semSEMAPHORE_QUEUE_ITEM_LENGTH (C macro), 1079
- shared_stack_function (C++ type), 1161
- shutdown_handler_t (C++ type), 1176
- sigmadelta_channel_t (C++ type), 734
- sigmadelta_config (C++ function), 733
- sigmadelta_config_t (C++ class), 734
- sigmadelta_config_t::channel (C++ member), 734
- sigmadelta_config_t::sigmadelta_duty (C++ member), 734
- sigmadelta_config_t::sigmadelta_gpio (C++ member), 734
- sigmadelta_config_t::sigmadelta_prescale (C++ member), 734
- sigmadelta_port_t (C++ type), 734
- sigmadelta_set_duty (C++ function), 733
- sigmadelta_set_pin (C++ function), 733
- sigmadelta_set_prescale (C++ function), 733
- slave_transaction_cb_t (C++ type), 759
- smartconfig_event_got_ssid_pswd_t (C++ class), 523
- smartconfig_event_got_ssid_pswd_t::bssid (C++ member), 523
- smartconfig_event_got_ssid_pswd_t::bssid_set (C++ member), 523
- smartconfig_event_got_ssid_pswd_t::cellphone_ip (C++ member), 523
- smartconfig_event_got_ssid_pswd_t::password (C++ member), 523
- smartconfig_event_got_ssid_pswd_t::ssid (C++ member), 523
- smartconfig_event_got_ssid_pswd_t::token (C++ member), 523
- smartconfig_event_got_ssid_pswd_t::type (C++ member), 523
- smartconfig_event_t (C++ enum), 524
- SMARTCONFIG_START_CONFIG_DEFAULT (C macro), 523
- smartconfig_start_config_t (C++ class), 523
- smartconfig_start_config_t::enable_log (C++ member), 523
- smartconfig_type_t (C++ enum), 524
- sntp_get_sync_interval (C++ function), 1209
- sntp_get_sync_mode (C++ function), 1209
- sntp_get_sync_status (C++ function), 1209
- sntp_restart (C++ function), 1210
- sntp_set_sync_interval (C++ function), 1209
- sntp_set_sync_mode (C++ function), 1209
- sntp_set_sync_status (C++ function), 1209
- sntp_set_time_sync_notification_cb (C++ function), 1209
- SNTP_SYNC_MODE_IMMED (C++ enumerator), 1210
- SNTP_SYNC_MODE_SMOOTH (C++ enumerator), 1210
- sntp_sync_mode_t (C++ enum), 1210
- SNTP_SYNC_STATUS_COMPLETED (C++ enumerator), 1210
- SNTP_SYNC_STATUS_IN_PROGRESS (C++ enumerator), 1210
- SNTP_SYNC_STATUS_RESET (C++ enumerator), 1210
- sntp_sync_status_t (C++ enum), 1210
- sntp_sync_time (C++ function), 1209
- sntp_sync_time_cb_t (C++ type), 1210
- soc_has_cache_lock_bug (C++ function), 1017
- SPI1_HOST (C++ enumerator), 743

- SPI2_HOST (C++ enumerator), 743
 SPI3_HOST (C++ enumerator), 743
 spi_bus_add_device (C++ function), 745
 spi_bus_add_flash_device (C++ function), 967
 spi_bus_config_t (C++ class), 744
 spi_bus_config_t::flags (C++ member), 744
 spi_bus_config_t::intr_flags (C++ member), 744
 spi_bus_config_t::max_transfer_sz (C++ member), 744
 spi_bus_config_t::miso_io_num (C++ member), 744
 spi_bus_config_t::mosi_io_num (C++ member), 744
 spi_bus_config_t::quadhd_io_num (C++ member), 744
 spi_bus_config_t::quadwp_io_num (C++ member), 744
 spi_bus_config_t::sclk_io_num (C++ member), 744
 spi_bus_free (C++ function), 743
 spi_bus_initialize (C++ function), 743
 spi_bus_remove_device (C++ function), 746
 spi_bus_remove_flash_device (C++ function), 967
 spi_cal_clock (C++ function), 748
 SPI_DEVICE_3WIRE (C macro), 751
 spi_device_acquire_bus (C++ function), 747
 SPI_DEVICE_BIT_LSBFIRST (C macro), 751
 SPI_DEVICE_CLK_AS_CS (C macro), 751
 SPI_DEVICE_DDRCLK (C macro), 751
 spi_device_get_trans_result (C++ function), 746
 SPI_DEVICE_HALFDUPLEX (C macro), 751
 spi_device_handle_t (C++ type), 751
 spi_device_interface_config_t (C++ class), 748
 spi_device_interface_config_t::address_bits (C++ member), 749
 spi_device_interface_config_t::clock_speed_hz (C++ member), 749
 spi_device_interface_config_t::command_bits (C++ member), 748
 spi_device_interface_config_t::cs_ena_pos (C++ member), 749
 spi_device_interface_config_t::cs_ena_posn (C++ member), 749
 spi_device_interface_config_t::dummy_bits (C++ member), 749
 spi_device_interface_config_t::duty_cycle_pos (C++ member), 749
 spi_device_interface_config_t::flags (C++ member), 749
 spi_device_interface_config_t::input_delay_ns (C++ member), 749
 spi_device_interface_config_t::mode (C++ member), 749
 spi_device_interface_config_t::post_cb (C++ member), 749
 spi_device_interface_config_t::pre_cb (C++ member), 749
 spi_device_interface_config_t::queue_size (C++ member), 749
 spi_device_interface_config_t::spics_io_num (C++ member), 749
 SPI_DEVICE_NO_DUMMY (C macro), 751
 spi_device_polling_end (C++ function), 747
 spi_device_polling_start (C++ function), 747
 spi_device_polling_transmit (C++ function), 747
 SPI_DEVICE_POSITIVE_CS (C macro), 751
 spi_device_queue_trans (C++ function), 746
 spi_device_release_bus (C++ function), 748
 SPI_DEVICE_RXBIT_LSBFIRST (C macro), 751
 spi_device_transmit (C++ function), 746
 SPI_DEVICE_TXBIT_LSBFIRST (C macro), 750
 spi_flash_chip_t (C++ type), 972
 SPI_FLASH_DIO (C++ enumerator), 974
 SPI_FLASH_DOUT (C++ enumerator), 974
 SPI_FLASH_FASTRD (C++ enumerator), 974
 spi_flash_host_driver_t (C++ class), 972
 spi_flash_host_driver_t (C++ type), 973
 spi_flash_host_driver_t::common_command (C++ member), 972
 spi_flash_host_driver_t::configure_host_io_mode (C++ member), 973
 spi_flash_host_driver_t::dev_config (C++ member), 972
 spi_flash_host_driver_t::driver_data (C++ member), 972
 spi_flash_host_driver_t::erase_block (C++ member), 972
 spi_flash_host_driver_t::erase_chip (C++ member), 972
 spi_flash_host_driver_t::erase_sector (C++ member), 972
 spi_flash_host_driver_t::flush_cache (C++ member), 973
 spi_flash_host_driver_t::host_idle (C++ member), 973
 spi_flash_host_driver_t::max_read_bytes (C++ member), 973
 spi_flash_host_driver_t::max_write_bytes (C++ member), 973
 spi_flash_host_driver_t::poll_cmd_done (C++ member), 973
 spi_flash_host_driver_t::program_page (C++ member), 973
 spi_flash_host_driver_t::read (C++ member), 973
 spi_flash_host_driver_t::read_id (C++ member), 972
 spi_flash_host_driver_t::read_status (C++ member), 973

- [spi_flash_host_driver_t::set_write_protect](#) (C++ member), 758
 (C++ member), 973
- [spi_flash_host_driver_t::supports_direct_read](#) (C++ member), 758
 (C++ member), 973
- [spi_flash_host_driver_t::supports_direct_write](#) (C++ member), 758
 (C++ member), 973
- [SPI_FLASH_QIO](#) (C++ enumerator), 974
- [SPI_FLASH_QOUT](#) (C++ enumerator), 974
- [SPI_FLASH_READ_MODE_MAX](#) (C++ enumerator), 974
- [SPI_FLASH_READ_MODE_MIN](#) (C macro), 973
- [SPI_FLASH_SLOWRD](#) (C++ enumerator), 974
- [spi_flash_trans_t](#) (C++ class), 972
- [spi_flash_trans_t::address](#) (C++ member), 972
- [spi_flash_trans_t::address_bitlen](#) (C++ member), 972
- [spi_flash_trans_t::command](#) (C++ member), 972
- [spi_flash_trans_t::miso_data](#) (C++ member), 972
- [spi_flash_trans_t::miso_len](#) (C++ member), 972
- [spi_flash_trans_t::mosi_data](#) (C++ member), 972
- [spi_flash_trans_t::mosi_len](#) (C++ member), 972
- [spi_get_actual_clock](#) (C++ function), 748
- [spi_get_freq_limit](#) (C++ function), 748
- [spi_get_timing](#) (C++ function), 748
- [spi_host_device_t](#) (C++ enum), 743
- [SPI_MAX_DMA_LEN](#) (C macro), 744
- [SPI_SLAVE_BIT_LSBFIRST](#) (C macro), 758
- [spi_slave_free](#) (C++ function), 756
- [spi_slave_get_trans_result](#) (C++ function), 757
- [spi_slave_initialize](#) (C++ function), 756
- [spi_slave_interface_config_t](#) (C++ class), 758
- [spi_slave_interface_config_t::flags](#) (C++ member), 758
- [spi_slave_interface_config_t::mode](#) (C++ member), 758
- [spi_slave_interface_config_t::post_setup](#) (C++ member), 758
- [spi_slave_interface_config_t::post_transfer](#) (C++ member), 758
- [spi_slave_interface_config_t::queue_size](#) (C++ member), 758
- [spi_slave_interface_config_t::spics_io](#) (C++ member), 758
- [spi_slave_queue_trans](#) (C++ function), 757
- [SPI_SLAVE_RXBIT_LSBFIRST](#) (C macro), 758
- [spi_slave_transaction_t](#) (C++ class), 758
- [spi_slave_transaction_t](#) (C++ type), 759
- [spi_slave_transaction_t::length](#) (C++ member), 758
- [spi_slave_transaction_t::rx_buffer](#)
- [spi_slave_transaction_t::trans_len](#)
- [spi_slave_transaction_t::tx_buffer](#)
- [spi_slave_transaction_t::user](#) (C++ member), 758
- [spi_slave_transmit](#) (C++ function), 757
- [SPI_SLAVE_TXBIT_LSBFIRST](#) (C macro), 758
- [SPI_SWAP_DATA_RX](#) (C macro), 744
- [SPI_SWAP_DATA_TX](#) (C macro), 744
- [SPI_TRANS_MODE_DIO](#) (C macro), 751
- [SPI_TRANS_MODE_DIOQIO_ADDR](#) (C macro), 751
- [SPI_TRANS_MODE_QIO](#) (C macro), 751
- [SPI_TRANS_SET_CD](#) (C macro), 751
- [SPI_TRANS_USE_RXDATA](#) (C macro), 751
- [SPI_TRANS_USE_TXDATA](#) (C macro), 751
- [SPI_TRANS_VARIABLE_ADDR](#) (C macro), 751
- [SPI_TRANS_VARIABLE_CMD](#) (C macro), 751
- [SPI_TRANS_VARIABLE_DUMMY](#) (C macro), 751
- [spi_transaction_ext_t](#) (C++ class), 750
- [spi_transaction_ext_t::address_bits](#) (C++ member), 750
- [spi_transaction_ext_t::base](#) (C++ member), 750
- [spi_transaction_ext_t::command_bits](#) (C++ member), 750
- [spi_transaction_ext_t::dummy_bits](#) (C++ member), 750
- [spi_transaction_t](#) (C++ class), 749
- [spi_transaction_t](#) (C++ type), 751
- [spi_transaction_t::addr](#) (C++ member), 750
- [spi_transaction_t::cmd](#) (C++ member), 749
- [spi_transaction_t::flags](#) (C++ member), 749
- [spi_transaction_t::length](#) (C++ member), 750
- [spi_transaction_t::rx_buffer](#) (C++ member), 750
- [spi_transaction_t::rx_data](#) (C++ member), 750
- [spi_transaction_t::rxlength](#) (C++ member), 750
- [spi_transaction_t::tx_buffer](#) (C++ member), 750
- [spi_transaction_t::tx_data](#) (C++ member), 750
- [spi_transaction_t::user](#) (C++ member), 750
- [SPICOMMON_BUSFLAG_DUAL](#) (C macro), 745
- [SPICOMMON_BUSFLAG_IOMUX_PINS](#) (C macro), 745
- [SPICOMMON_BUSFLAG_MASTER](#) (C macro), 745
- [SPICOMMON_BUSFLAG_MISO](#) (C macro), 745
- [SPICOMMON_BUSFLAG_MOSI](#) (C macro), 745
- [SPICOMMON_BUSFLAG_NATIVE_PINS](#) (C macro), 745
- [SPICOMMON_BUSFLAG_QUAD](#) (C macro), 745
- [SPICOMMON_BUSFLAG_SCLK](#) (C macro), 745

- SPICOMMON_BUSFLAG_SLAVE (*C macro*), 745
- SPICOMMON_BUSFLAG_WPHD (*C macro*), 745
- StaticRingbuffer_t (*C++ type*), 1129
- system_event_ap_probe_req_rx_t (*C++ type*), 1035
- SYSTEM_EVENT_AP_PROBEREQRECVED (*C++ enumerator*), 1036
- SYSTEM_EVENT_AP_STA_GOT_IP6 (*C macro*), 1034
- SYSTEM_EVENT_AP_STACONNECTED (*C++ enumerator*), 1036
- system_event_ap_staconnected_t (*C++ type*), 1035
- SYSTEM_EVENT_AP_STADISCONNECTED (*C++ enumerator*), 1036
- system_event_ap_stadisconnected_t (*C++ type*), 1035
- SYSTEM_EVENT_AP_STAIPASSIGNED (*C++ enumerator*), 1036
- system_event_ap_staipassigned_t (*C++ type*), 1035
- SYSTEM_EVENT_AP_START (*C++ enumerator*), 1036
- SYSTEM_EVENT_AP_STOP (*C++ enumerator*), 1036
- system_event_cb_t (*C++ type*), 1035
- SYSTEM_EVENT_ETH_CONNECTED (*C++ enumerator*), 1036
- SYSTEM_EVENT_ETH_DISCONNECTED (*C++ enumerator*), 1036
- SYSTEM_EVENT_ETH_GOT_IP (*C++ enumerator*), 1036
- SYSTEM_EVENT_ETH_START (*C++ enumerator*), 1036
- SYSTEM_EVENT_ETH_STOP (*C++ enumerator*), 1036
- SYSTEM_EVENT_GOT_IP6 (*C++ enumerator*), 1036
- system_event_got_ip6_t (*C++ type*), 1035
- system_event_handler_t (*C++ type*), 1035
- system_event_id_t (*C++ enum*), 1035
- system_event_info_t (*C++ union*), 1033
- system_event_info_t::ap_probereqrecv (*C++ member*), 1034
- system_event_info_t::ap_staipassigned (*C++ member*), 1034
- system_event_info_t::auth_change (*C++ member*), 1034
- system_event_info_t::connected (*C++ member*), 1034
- system_event_info_t::disconnected (*C++ member*), 1034
- system_event_info_t::got_ip (*C++ member*), 1034
- system_event_info_t::got_ip6 (*C++ member*), 1034
- system_event_info_t::scan_done (*C++ member*), 1034
- system_event_info_t::sta_connected (*C++ member*), 1034
- system_event_info_t::sta_disconnected (*C++ member*), 1034
- system_event_info_t::sta_er_fail_reason (*C++ member*), 1034
- system_event_info_t::sta_er_pin (*C++ member*), 1034
- system_event_info_t::sta_er_success (*C++ member*), 1034
- SYSTEM_EVENT_MAX (*C++ enumerator*), 1036
- SYSTEM_EVENT_SCAN_DONE (*C++ enumerator*), 1035
- SYSTEM_EVENT_STA_AUTHMODE_CHANGE (*C++ enumerator*), 1036
- system_event_sta_authmode_change_t (*C++ type*), 1035
- SYSTEM_EVENT_STA_BEACON_TIMEOUT (*C++ enumerator*), 1036
- SYSTEM_EVENT_STA_CONNECTED (*C++ enumerator*), 1035
- system_event_sta_connected_t (*C++ type*), 1034
- SYSTEM_EVENT_STA_DISCONNECTED (*C++ enumerator*), 1036
- system_event_sta_disconnected_t (*C++ type*), 1035
- SYSTEM_EVENT_STA_GOT_IP (*C++ enumerator*), 1036
- system_event_sta_got_ip_t (*C++ type*), 1035
- SYSTEM_EVENT_STA_LOST_IP (*C++ enumerator*), 1036
- system_event_sta_scan_done_t (*C++ type*), 1034
- SYSTEM_EVENT_STA_START (*C++ enumerator*), 1035
- SYSTEM_EVENT_STA_STOP (*C++ enumerator*), 1035
- SYSTEM_EVENT_STA_WPS_ER_FAILED (*C++ enumerator*), 1036
- SYSTEM_EVENT_STA_WPS_ER_PBC_OVERLAP (*C++ enumerator*), 1036
- SYSTEM_EVENT_STA_WPS_ER_PIN (*C++ enumerator*), 1036
- system_event_sta_wps_er_pin_t (*C++ type*), 1035
- SYSTEM_EVENT_STA_WPS_ER_SUCCESS (*C++ enumerator*), 1036
- system_event_sta_wps_er_success_t (*C++ type*), 1035
- SYSTEM_EVENT_STA_WPS_ER_TIMEOUT (*C++ enumerator*), 1036
- system_event_sta_wps_fail_reason_t (*C++ type*), 1034
- system_event_t (*C++ class*), 1034
- system_event_t::event_id (*C++ member*), 1034
- system_event_t::event_info (*C++ member*), 1034
- SYSTEM_EVENT_WIFI_READY (*C++ enumerator*),

1035

T

- taskDISABLE_INTERRUPTS (*C macro*), 1057
- taskENABLE_INTERRUPTS (*C macro*), 1057
- taskENTER_CRITICAL (*C macro*), 1057
- taskENTER_CRITICAL_ISR (*C macro*), 1057
- taskEXIT_CRITICAL (*C macro*), 1057
- taskEXIT_CRITICAL_ISR (*C macro*), 1057
- TaskHandle_t (*C++ type*), 1058
- TaskHookFunction_t (*C++ type*), 1058
- taskSCHEDULER_NOT_STARTED (*C macro*), 1057
- taskSCHEDULER_RUNNING (*C macro*), 1057
- taskSCHEDULER_SUSPENDED (*C macro*), 1057
- TaskSnapshot_t (*C++ type*), 1058
- TaskStatus_t (*C++ type*), 1058
- taskYIELD (*C macro*), 1057
- temp_sensor_config_t (*C++ class*), 760
- temp_sensor_config_t::clk_div (*C++ member*), 760
- temp_sensor_config_t::dac_offset (*C++ member*), 760
- temp_sensor_dac_offset_t (*C++ enum*), 760
- temp_sensor_get_config (*C++ function*), 759
- temp_sensor_read_celsius (*C++ function*), 760
- temp_sensor_read_raw (*C++ function*), 760
- temp_sensor_set_config (*C++ function*), 759
- temp_sensor_start (*C++ function*), 759
- temp_sensor_stop (*C++ function*), 759
- TIMER_0 (*C++ enumerator*), 769
- TIMER_1 (*C++ enumerator*), 769
- TIMER_ALARM_DIS (*C++ enumerator*), 770
- TIMER_ALARM_EN (*C++ enumerator*), 770
- TIMER_ALARM_MAX (*C++ enumerator*), 770
- timer_alarm_t (*C++ enum*), 770
- TIMER_AUTORELOAD_DIS (*C++ enumerator*), 771
- TIMER_AUTORELOAD_EN (*C++ enumerator*), 771
- TIMER_AUTORELOAD_MAX (*C++ enumerator*), 771
- timer_autoreload_t (*C++ enum*), 771
- TIMER_BASE_CLK (*C macro*), 768
- timer_config_t (*C++ class*), 769
- timer_config_t::alarm_en (*C++ member*), 769
- timer_config_t::auto_reload (*C++ member*), 769
- timer_config_t::counter_dir (*C++ member*), 769
- timer_config_t::counter_en (*C++ member*), 769
- timer_config_t::divider (*C++ member*), 769
- timer_config_t::intr_type (*C++ member*), 769
- timer_count_dir_t (*C++ enum*), 770
- TIMER_COUNT_DOWN (*C++ enumerator*), 770
- TIMER_COUNT_MAX (*C++ enumerator*), 770
- TIMER_COUNT_UP (*C++ enumerator*), 770
- timer_deinit (*C++ function*), 766
- timer_disable_intr (*C++ function*), 766
- timer_enable_intr (*C++ function*), 766
- timer_get_alarm_value (*C++ function*), 764
- timer_get_config (*C++ function*), 766
- timer_get_counter_time_sec (*C++ function*), 763
- timer_get_counter_value (*C++ function*), 763
- TIMER_GROUP_0 (*C++ enumerator*), 769
- TIMER_GROUP_1 (*C++ enumerator*), 769
- timer_group_clr_intr_sta_in_isr (*C++ function*), 768
- timer_group_clr_intr_status_in_isr (*C++ function*), 767
- timer_group_enable_alarm_in_isr (*C++ function*), 767
- timer_group_get_auto_reload_in_isr (*C++ function*), 768
- timer_group_get_counter_value_in_isr (*C++ function*), 767
- timer_group_get_intr_status_in_isr (*C++ function*), 768
- timer_group_intr_clr_in_isr (*C++ function*), 767
- timer_group_intr_disable (*C++ function*), 766
- timer_group_intr_enable (*C++ function*), 766
- timer_group_intr_get_in_isr (*C++ function*), 768
- TIMER_GROUP_MAX (*C++ enumerator*), 769
- timer_group_set_alarm_value_in_isr (*C++ function*), 767
- timer_group_set_counter_enable_in_isr (*C++ function*), 767
- timer_group_t (*C++ enum*), 769
- timer_idx_t (*C++ enum*), 769
- timer_init (*C++ function*), 765
- TIMER_INTR_LEVEL (*C++ enumerator*), 771
- TIMER_INTR_MAX (*C++ enumerator*), 771
- timer_intr_mode_t (*C++ enum*), 770
- TIMER_INTR_NONE (*C++ enumerator*), 770
- timer_intr_t (*C++ enum*), 770
- TIMER_INTR_T0 (*C++ enumerator*), 770
- TIMER_INTR_T1 (*C++ enumerator*), 770
- TIMER_INTR_WDT (*C++ enumerator*), 770
- timer_isr_callback_add (*C++ function*), 764
- timer_isr_callback_remove (*C++ function*), 765
- timer_isr_handle_t (*C++ type*), 769
- timer_isr_register (*C++ function*), 765
- timer_isr_t (*C++ type*), 768
- TIMER_MAX (*C++ enumerator*), 769
- TIMER_PAUSE (*C++ enumerator*), 770
- timer_pause (*C++ function*), 763
- timer_set_alarm (*C++ function*), 764
- timer_set_alarm_value (*C++ function*), 764
- timer_set_auto_reload (*C++ function*), 764
- timer_set_counter_mode (*C++ function*), 763
- timer_set_counter_value (*C++ function*), 763

- [timer_set_divider \(C++ function\), 764](#)
[timer_spinlock_give \(C++ function\), 768](#)
[timer_spinlock_take \(C++ function\), 768](#)
[TIMER_START \(C++ enumerator\), 770](#)
[timer_start \(C++ function\), 763](#)
[timer_start_t \(C++ enum\), 770](#)
[timer_wdt_behavior_t \(C++ enum\), 770](#)
[TIMER_WDT_INT \(C++ enumerator\), 770](#)
[TIMER_WDT_OFF \(C++ enumerator\), 770](#)
[TIMER_WDT_RESET_CPU \(C++ enumerator\), 770](#)
[TIMER_WDT_RESET_SYSTEM \(C++ enumerator\), 770](#)
[TimerCallbackFunction_t \(C++ type\), 1107](#)
[TimerHandle_t \(C++ type\), 1107](#)
[tls_keep_alive_cfg \(C++ class\), 816](#)
[tls_keep_alive_cfg::keep_alive_count \(C++ member\), 816](#)
[tls_keep_alive_cfg::keep_alive_enable \(C++ member\), 816](#)
[tls_keep_alive_cfg::keep_alive_idle \(C++ member\), 816](#)
[tls_keep_alive_cfg::keep_alive_interval \(C++ member\), 816](#)
[tls_keep_alive_cfg_t \(C++ type\), 819](#)
[TlsDeleteCallbackFunction_t \(C++ type\), 1058](#)
[tmrCOMMAND_CHANGE_PERIOD \(C macro\), 1098](#)
[tmrCOMMAND_CHANGE_PERIOD_FROM_ISR \(C macro\), 1098](#)
[tmrCOMMAND_DELETE \(C macro\), 1098](#)
[tmrCOMMAND_EXECUTE_CALLBACK \(C macro\), 1098](#)
[tmrCOMMAND_EXECUTE_CALLBACK_FROM_ISR \(C macro\), 1098](#)
[tmrCOMMAND_RESET \(C macro\), 1098](#)
[tmrCOMMAND_RESET_FROM_ISR \(C macro\), 1098](#)
[tmrCOMMAND_START \(C macro\), 1098](#)
[tmrCOMMAND_START_DONT_TRACE \(C macro\), 1098](#)
[tmrCOMMAND_START_FROM_ISR \(C macro\), 1098](#)
[tmrCOMMAND_STOP \(C macro\), 1098](#)
[tmrCOMMAND_STOP_FROM_ISR \(C macro\), 1098](#)
[tmrFIRST_FROM_ISR_COMMAND \(C macro\), 1098](#)
[touch_cnt_slope_t \(C++ enum\), 776](#)
[TOUCH_FSM_MODE_MAX \(C++ enumerator\), 777](#)
[TOUCH_FSM_MODE_SW \(C++ enumerator\), 777](#)
[touch_fsm_mode_t \(C++ enum\), 777](#)
[TOUCH_FSM_MODE_TIMER \(C++ enumerator\), 777](#)
[touch_high_volt_t \(C++ enum\), 775](#)
[TOUCH_HVOLT_2V4 \(C++ enumerator\), 775](#)
[TOUCH_HVOLT_2V5 \(C++ enumerator\), 775](#)
[TOUCH_HVOLT_2V6 \(C++ enumerator\), 775](#)
[TOUCH_HVOLT_2V7 \(C++ enumerator\), 776](#)
[TOUCH_HVOLT_ATTEN_0V \(C++ enumerator\), 776](#)
[TOUCH_HVOLT_ATTEN_0V5 \(C++ enumerator\), 776](#)
[TOUCH_HVOLT_ATTEN_1V \(C++ enumerator\), 776](#)
[TOUCH_HVOLT_ATTEN_1V5 \(C++ enumerator\), 776](#)
[TOUCH_HVOLT_ATTEN_KEEP \(C++ enumerator\), 776](#)
[TOUCH_HVOLT_ATTEN_MAX \(C++ enumerator\), 776](#)
[TOUCH_HVOLT_KEEP \(C++ enumerator\), 775](#)
[TOUCH_HVOLT_MAX \(C++ enumerator\), 776](#)
[touch_low_volt_t \(C++ enum\), 776](#)
[TOUCH_LVOLT_0V5 \(C++ enumerator\), 776](#)
[TOUCH_LVOLT_0V6 \(C++ enumerator\), 776](#)
[TOUCH_LVOLT_0V7 \(C++ enumerator\), 776](#)
[TOUCH_LVOLT_0V8 \(C++ enumerator\), 776](#)
[TOUCH_LVOLT_KEEP \(C++ enumerator\), 776](#)
[TOUCH_LVOLT_MAX \(C++ enumerator\), 776](#)
[TOUCH_PAD_ATTEN_VOLTAGE_THRESHOLD \(C macro\), 775](#)
[TOUCH_PAD_BIT_MASK_MAX \(C macro\), 775](#)
[TOUCH_PAD_GPIO0_CHANNEL \(C macro\), 774](#)
[TOUCH_PAD_GPIO12_CHANNEL \(C macro\), 774](#)
[TOUCH_PAD_GPIO13_CHANNEL \(C macro\), 774](#)
[TOUCH_PAD_GPIO14_CHANNEL \(C macro\), 774](#)
[TOUCH_PAD_GPIO15_CHANNEL \(C macro\), 774](#)
[TOUCH_PAD_GPIO27_CHANNEL \(C macro\), 774](#)
[TOUCH_PAD_GPIO2_CHANNEL \(C macro\), 774](#)
[TOUCH_PAD_GPIO32_CHANNEL \(C macro\), 774](#)
[TOUCH_PAD_GPIO33_CHANNEL \(C macro\), 774](#)
[TOUCH_PAD_GPIO4_CHANNEL \(C macro\), 774](#)
[TOUCH_PAD_HIGH_VOLTAGE_THRESHOLD \(C macro\), 775](#)
[TOUCH_PAD_INACTIVE_CONNECT_DEFAULT \(C macro\), 775](#)
[TOUCH_PAD_LOW_VOLTAGE_THRESHOLD \(C macro\), 775](#)
[TOUCH_PAD_MAX \(C++ enumerator\), 775](#)
[TOUCH_PAD_NUM0 \(C++ enumerator\), 775](#)
[TOUCH_PAD_NUM0_GPIO_NUM \(C macro\), 774](#)
[TOUCH_PAD_NUM1 \(C++ enumerator\), 775](#)
[TOUCH_PAD_NUM1_GPIO_NUM \(C macro\), 774](#)
[TOUCH_PAD_NUM2 \(C++ enumerator\), 775](#)
[TOUCH_PAD_NUM2_GPIO_NUM \(C macro\), 774](#)
[TOUCH_PAD_NUM3 \(C++ enumerator\), 775](#)
[TOUCH_PAD_NUM3_GPIO_NUM \(C macro\), 774](#)
[TOUCH_PAD_NUM4 \(C++ enumerator\), 775](#)
[TOUCH_PAD_NUM4_GPIO_NUM \(C macro\), 774](#)
[TOUCH_PAD_NUM5 \(C++ enumerator\), 775](#)
[TOUCH_PAD_NUM5_GPIO_NUM \(C macro\), 774](#)
[TOUCH_PAD_NUM6 \(C++ enumerator\), 775](#)
[TOUCH_PAD_NUM6_GPIO_NUM \(C macro\), 774](#)
[TOUCH_PAD_NUM7 \(C++ enumerator\), 775](#)
[TOUCH_PAD_NUM7_GPIO_NUM \(C macro\), 774](#)
[TOUCH_PAD_NUM8 \(C++ enumerator\), 775](#)
[TOUCH_PAD_NUM8_GPIO_NUM \(C macro\), 774](#)
[TOUCH_PAD_NUM9 \(C++ enumerator\), 775](#)
[TOUCH_PAD_NUM9_GPIO_NUM \(C macro\), 774](#)
[TOUCH_PAD_SLOPE_0 \(C++ enumerator\), 776](#)
[TOUCH_PAD_SLOPE_1 \(C++ enumerator\), 776](#)
[TOUCH_PAD_SLOPE_2 \(C++ enumerator\), 776](#)
[TOUCH_PAD_SLOPE_3 \(C++ enumerator\), 776](#)
[TOUCH_PAD_SLOPE_4 \(C++ enumerator\), 776](#)
[TOUCH_PAD_SLOPE_5 \(C++ enumerator\), 776](#)

- TOUCH_PAD_SLOPE_6 (C++ enumerator), 776
 TOUCH_PAD_SLOPE_7 (C++ enumerator), 777
 TOUCH_PAD_SLOPE_DEFAULT (C macro), 775
 TOUCH_PAD_SLOPE_MAX (C++ enumerator), 777
 touch_pad_t (C++ enum), 775
 TOUCH_PAD_THRESHOLD_MAX (C macro), 775
 TOUCH_PAD_TIE_OPT_DEFAULT (C macro), 775
 TOUCH_PAD_TIE_OPT_HIGH (C++ enumerator), 777
 TOUCH_PAD_TIE_OPT_LOW (C++ enumerator), 777
 TOUCH_PAD_TIE_OPT_MAX (C++ enumerator), 777
 touch_tie_opt_t (C++ enum), 777
 TOUCH_TRIGGER_ABOVE (C++ enumerator), 777
 TOUCH_TRIGGER_BELOW (C++ enumerator), 777
 TOUCH_TRIGGER_MAX (C++ enumerator), 777
 touch_trigger_mode_t (C++ enum), 777
 TOUCH_TRIGGER_SOURCE_BOTH (C++ enumerator), 777
 TOUCH_TRIGGER_SOURCE_MAX (C++ enumerator), 777
 TOUCH_TRIGGER_SOURCE_SET1 (C++ enumerator), 777
 touch_trigger_src_t (C++ enum), 777
 touch_volt_atten_t (C++ enum), 776
 transaction_cb_t (C++ type), 751
 TSENS_CONFIG_DEFAULT (C macro), 760
 TSENS_DAC_DEFAULT (C++ enumerator), 760
 TSENS_DAC_L0 (C++ enumerator), 760
 TSENS_DAC_L1 (C++ enumerator), 760
 TSENS_DAC_L2 (C++ enumerator), 760
 TSENS_DAC_L3 (C++ enumerator), 760
 TSENS_DAC_L4 (C++ enumerator), 760
 TSENS_DAC_MAX (C++ enumerator), 760
 tskIDLE_PRIORITY (C macro), 1057
 tskKERNEL_VERSION_BUILD (C macro), 1057
 tskKERNEL_VERSION_MAJOR (C macro), 1057
 tskKERNEL_VERSION_MINOR (C macro), 1057
 tskKERNEL_VERSION_NUMBER (C macro), 1057
 tskNO_AFFINITY (C macro), 1057
- ## U
- uart_at_cmd_t (C++ class), 795
 uart_at_cmd_t::char_num (C++ member), 795
 uart_at_cmd_t::cmd_char (C++ member), 795
 uart_at_cmd_t::gap_tout (C++ member), 795
 uart_at_cmd_t::post_idle (C++ member), 795
 uart_at_cmd_t::pre_idle (C++ member), 795
 UART_BREAK (C++ enumerator), 795
 UART_BUFFER_FULL (C++ enumerator), 795
 uart_clear_intr_status (C++ function), 786
 uart_config_t (C++ class), 796
 uart_config_t::baud_rate (C++ member), 796
 uart_config_t::data_bits (C++ member), 796
 uart_config_t::flow_ctrl (C++ member), 796
 uart_config_t::parity (C++ member), 796
 uart_config_t::rx_flow_ctrl_thresh (C++ member), 796
 uart_config_t::source_clk (C++ member), 796
 uart_config_t::stop_bits (C++ member), 796
 UART_CTS_GPIO19_DIRECT_CHANNEL (C macro), 799
 UART_CTS_GPIO6_DIRECT_CHANNEL (C macro), 799
 UART_CTS_GPIO8_DIRECT_CHANNEL (C macro), 799
 UART_DATA (C++ enumerator), 794
 UART_DATA_5_BITS (C++ enumerator), 796
 UART_DATA_6_BITS (C++ enumerator), 797
 UART_DATA_7_BITS (C++ enumerator), 797
 UART_DATA_8_BITS (C++ enumerator), 797
 UART_DATA_BITS_MAX (C++ enumerator), 797
 UART_DATA_BREAK (C++ enumerator), 795
 uart_disable_intr_mask (C++ function), 786
 uart_disable_pattern_det_intr (C++ function), 790
 uart_disable_rx_intr (C++ function), 787
 uart_disable_tx_intr (C++ function), 787
 uart_driver_delete (C++ function), 784
 uart_driver_install (C++ function), 783
 uart_enable_intr_mask (C++ function), 786
 uart_enable_pattern_det_baud_intr (C++ function), 790
 uart_enable_rx_intr (C++ function), 786
 uart_enable_tx_intr (C++ function), 787
 UART_EVENT_MAX (C++ enumerator), 795
 uart_event_t (C++ class), 794
 uart_event_t::size (C++ member), 794
 uart_event_t::timeout_flag (C++ member), 794
 uart_event_t::type (C++ member), 794
 uart_event_type_t (C++ enum), 794
 UART_FIFO_OVF (C++ enumerator), 795
 uart_flush (C++ function), 790
 uart_flush_input (C++ function), 790
 UART_FRAME_ERR (C++ enumerator), 795
 uart_get_baudrate (C++ function), 785
 uart_get_buffered_data_len (C++ function), 790
 uart_get_collision_flag (C++ function), 792
 uart_get_hw_flow_ctrl (C++ function), 786
 uart_get_parity (C++ function), 785
 uart_get_stop_bits (C++ function), 784
 uart_get_wakeup_threshold (C++ function), 793
 uart_get_word_length (C++ function), 784
 UART_GPIO10_DIRECT_CHANNEL (C macro), 799
 UART_GPIO11_DIRECT_CHANNEL (C macro), 799
 UART_GPIO16_DIRECT_CHANNEL (C macro), 799
 UART_GPIO17_DIRECT_CHANNEL (C macro), 799
 UART_GPIO19_DIRECT_CHANNEL (C macro), 798

- UART_GPIO1_DIRECT_CHANNEL (*C macro*), 798
 UART_GPIO22_DIRECT_CHANNEL (*C macro*), 799
 UART_GPIO3_DIRECT_CHANNEL (*C macro*), 798
 UART_GPIO6_DIRECT_CHANNEL (*C macro*), 799
 UART_GPIO7_DIRECT_CHANNEL (*C macro*), 799
 UART_GPIO8_DIRECT_CHANNEL (*C macro*), 799
 UART_GPIO9_DIRECT_CHANNEL (*C macro*), 799
 uart_hw_flowcontrol_t (*C++ enum*), 797
 UART_HW_FLOWCTRL_CTS (*C++ enumerator*), 797
 UART_HW_FLOWCTRL_CTS_RTS (*C++ enumerator*), 797
 UART_HW_FLOWCTRL_DISABLE (*C++ enumerator*), 797
 UART_HW_FLOWCTRL_MAX (*C++ enumerator*), 797
 UART_HW_FLOWCTRL_RTS (*C++ enumerator*), 797
 uart_intr_config (*C++ function*), 788
 uart_intr_config_t (*C++ class*), 794
 uart_intr_config_t::intr_enable_mask (*C++ member*), 794
 uart_intr_config_t::rx_timeout_thresh (*C++ member*), 794
 uart_intr_config_t::rxfifo_full_thresh (*C++ member*), 794
 uart_intr_config_t::txfifo_empty_intr_thresh (*C++ member*), 794
 uart_is_driver_installed (*C++ function*), 784
 uart_isr_free (*C++ function*), 787
 uart_isr_handle_t (*C++ type*), 794
 uart_isr_register (*C++ function*), 787
 UART_MODE_IRDA (*C++ enumerator*), 796
 UART_MODE_RS485_APP_CTRL (*C++ enumerator*), 796
 UART_MODE_RS485_COLLISION_DETECT (*C++ enumerator*), 796
 UART_MODE_RS485_HALF_DUPLEX (*C++ enumerator*), 796
 uart_mode_t (*C++ enum*), 796
 UART_MODE_UART (*C++ enumerator*), 796
 UART_NUM_0 (*C macro*), 794
 UART_NUM_0_CTS_DIRECT_GPIO_NUM (*C macro*), 799
 UART_NUM_0_RTS_DIRECT_GPIO_NUM (*C macro*), 799
 UART_NUM_0_RXD_DIRECT_GPIO_NUM (*C macro*), 798
 UART_NUM_0_TXD_DIRECT_GPIO_NUM (*C macro*), 798
 UART_NUM_1 (*C macro*), 794
 UART_NUM_1_CTS_DIRECT_GPIO_NUM (*C macro*), 799
 UART_NUM_1_RTS_DIRECT_GPIO_NUM (*C macro*), 799
 UART_NUM_1_RXD_DIRECT_GPIO_NUM (*C macro*), 799
 UART_NUM_1_TXD_DIRECT_GPIO_NUM (*C macro*), 799
 UART_NUM_2_CTS_DIRECT_GPIO_NUM (*C macro*), 799
 UART_NUM_2_RTS_DIRECT_GPIO_NUM (*C macro*), 799
 UART_NUM_2_RXD_DIRECT_GPIO_NUM (*C macro*), 799
 UART_NUM_2_TXD_DIRECT_GPIO_NUM (*C macro*), 799
 UART_NUM_MAX (*C macro*), 794
 uart_param_config (*C++ function*), 788
 UART_PARITY_DISABLE (*C++ enumerator*), 797
 UART_PARITY_ERR (*C++ enumerator*), 795
 UART_PARITY_EVEN (*C++ enumerator*), 797
 UART_PARITY_ODD (*C++ enumerator*), 797
 uart_parity_t (*C++ enum*), 797
 UART_PATTERN_DET (*C++ enumerator*), 795
 uart_pattern_get_pos (*C++ function*), 791
 uart_pattern_pop_pos (*C++ function*), 791
 uart_pattern_queue_reset (*C++ function*), 791
 UART_PIN_NO_CHANGE (*C macro*), 794
 uart_port_t (*C++ type*), 796
 uart_read_bytes (*C++ function*), 789
 UART_RTS_GPIO11_DIRECT_CHANNEL (*C macro*), 799
 UART_RTS_GPIO22_DIRECT_CHANNEL (*C macro*), 799
 UART_RTS_GPIO7_DIRECT_CHANNEL (*C macro*), 799
 UART_RXD_GPIO16_DIRECT_CHANNEL (*C macro*), 799
 UART_RXD_GPIO3_DIRECT_CHANNEL (*C macro*), 799
 UART_RXD_GPIO9_DIRECT_CHANNEL (*C macro*), 799
 UART_SCLK_APB (*C++ enumerator*), 798
 UART_SCLK_REF_TICK (*C++ enumerator*), 798
 uart_sclk_t (*C++ enum*), 798
 uart_set_always_rx_timeout (*C++ function*), 793
 uart_set_baudrate (*C++ function*), 785
 uart_set_dtr (*C++ function*), 788
 uart_set_hw_flow_ctrl (*C++ function*), 785
 uart_set_line_inverse (*C++ function*), 785
 uart_set_loop_back (*C++ function*), 793
 uart_set_mode (*C++ function*), 791
 uart_set_parity (*C++ function*), 785
 uart_set_pin (*C++ function*), 787
 uart_set_rts (*C++ function*), 788
 uart_set_rx_full_threshold (*C++ function*), 792
 uart_set_rx_timeout (*C++ function*), 792
 uart_set_stop_bits (*C++ function*), 784
 uart_set_sw_flow_ctrl (*C++ function*), 786
 uart_set_tx_empty_threshold (*C++ function*), 792
 uart_set_tx_idle_num (*C++ function*), 788
 uart_set_wakeup_threshold (*C++ function*), 792

- uart_set_word_length (C++ function), 784
 - UART_SIGNAL_CTS_INV (C++ enumerator), 798
 - UART_SIGNAL_DSR_INV (C++ enumerator), 798
 - UART_SIGNAL_DTR_INV (C++ enumerator), 798
 - uart_signal_inv_t (C++ enum), 797
 - UART_SIGNAL_IRDA_RX_INV (C++ enumerator), 797
 - UART_SIGNAL_IRDA_TX_INV (C++ enumerator), 797
 - UART_SIGNAL_RTS_INV (C++ enumerator), 798
 - UART_SIGNAL_RXD_INV (C++ enumerator), 798
 - UART_SIGNAL_TXD_INV (C++ enumerator), 798
 - UART_STOP_BITS_1 (C++ enumerator), 797
 - UART_STOP_BITS_1_5 (C++ enumerator), 797
 - UART_STOP_BITS_2 (C++ enumerator), 797
 - UART_STOP_BITS_MAX (C++ enumerator), 797
 - uart_stop_bits_t (C++ enum), 797
 - uart_sw_flowctrl_t (C++ class), 795
 - uart_sw_flowctrl_t::xoff_char (C++ member), 795
 - uart_sw_flowctrl_t::xoff_thrd (C++ member), 796
 - uart_sw_flowctrl_t::xon_char (C++ member), 795
 - uart_sw_flowctrl_t::xon_thrd (C++ member), 795
 - uart_tx_chars (C++ function), 789
 - UART_TXD_GPIO10_DIRECT_CHANNEL (C macro), 799
 - UART_TXD_GPIO17_DIRECT_CHANNEL (C macro), 799
 - UART_TXD_GPIO1_DIRECT_CHANNEL (C macro), 799
 - uart_wait_tx_done (C++ function), 789
 - uart_wait_tx_idle_polling (C++ function), 793
 - uart_word_length_t (C++ enum), 796
 - uart_write_bytes (C++ function), 789
 - uart_write_bytes_with_break (C++ function), 789
 - UINT16 (C++ type), 323
 - UINT32 (C++ type), 323
 - UINT64 (C++ type), 323
 - UINT8 (C++ type), 323
 - ulp_load_binary (C++ function), 1675, 1679
 - ulp_process_macros_and_load (C++ function), 1669
 - ulp_run (C++ function), 1669, 1675, 1679
 - ulp_set_wakeup_period (C++ function), 1676, 1680
 - ulTaskNotifyTake (C++ function), 1055
 - uxQueueMessagesWaiting (C++ function), 1063
 - uxQueueMessagesWaitingFromISR (C++ function), 1060
 - uxQueueSpacesAvailable (C++ function), 1063
 - uxSemaphoreGetCount (C macro), 1091
 - uxTaskGetNumberOfTasks (C++ function), 1047
 - uxTaskGetStackHighWaterMark (C++ function), 1047
 - uxTaskGetSystemState (C++ function), 1049
 - uxTaskPriorityGet (C++ function), 1042
 - uxTaskPriorityGetFromISR (C++ function), 1043
- ## V
- vendor_ie_data_t (C++ class), 508
 - vendor_ie_data_t::element_id (C++ member), 508
 - vendor_ie_data_t::length (C++ member), 508
 - vendor_ie_data_t::payload (C++ member), 508
 - vendor_ie_data_t::vendor_oui (C++ member), 508
 - vendor_ie_data_t::vendor_oui_type (C++ member), 508
 - vEventGroupDelete (C++ function), 1113
 - vprintf_like_t (C++ type), 1170
 - vQueueAddToRegistry (C++ function), 1064
 - vQueueDelete (C++ function), 1063
 - vQueueUnregisterQueue (C++ function), 1064
 - vRingbufferDelete (C++ function), 1127
 - vRingbufferGetInfo (C++ function), 1128
 - vRingbufferReturnItem (C++ function), 1127
 - vRingbufferReturnItemFromISR (C++ function), 1127
 - vSemaphoreDelete (C macro), 1091
 - vTaskDelay (C++ function), 1041
 - vTaskDelayUntil (C++ function), 1041
 - vTaskDelete (C++ function), 1040
 - vTaskGetRunTimeStats (C++ function), 1051
 - vTaskList (C++ function), 1051
 - vTaskNotifyGiveFromISR (C++ function), 1054
 - vTaskPrioritySet (C++ function), 1043
 - vTaskResume (C++ function), 1044
 - vTaskSetApplicationTaskTag (C++ function), 1048
 - vTaskSetThreadLocalStoragePointer (C++ function), 1048
 - vTaskSetThreadLocalStoragePointerAndDelCallback (C++ function), 1048
 - vTaskSuspend (C++ function), 1044
 - vTaskSuspendAll (C++ function), 1045
 - vTimerSetTimerID (C++ function), 1095
- ## W
- WEBSOCKET_EVENT_ANY (C++ enumerator), 883
 - WEBSOCKET_EVENT_CONNECTED (C++ enumerator), 883
 - WEBSOCKET_EVENT_DATA (C++ enumerator), 884
 - WEBSOCKET_EVENT_DISCONNECTED (C++ enumerator), 883
 - WEBSOCKET_EVENT_ERROR (C++ enumerator), 883
 - WEBSOCKET_EVENT_MAX (C++ enumerator), 884

- WEBSOCKET_TRANSPORT_OVER_SSL (C++ *enumerator*), 884
- WEBSOCKET_TRANSPORT_OVER_TCP (C++ *enumerator*), 884
- WEBSOCKET_TRANSPORT_UNKNOWN (C++ *enumerator*), 884
- wifi_active_scan_time_t (C++ *class*), 505
- wifi_active_scan_time_t::max (C++ *member*), 505
- wifi_active_scan_time_t::min (C++ *member*), 505
- WIFI_ALL_CHANNEL_SCAN (C++ *enumerator*), 518
- WIFI_AMPDU_RX_ENABLED (C *macro*), 503
- WIFI_AMPDU_TX_ENABLED (C *macro*), 503
- WIFI_ANT_ANT0 (C++ *enumerator*), 517
- WIFI_ANT_ANT1 (C++ *enumerator*), 517
- wifi_ant_config_t (C++ *class*), 511
- wifi_ant_config_t::enabled_ant0 (C++ *member*), 512
- wifi_ant_config_t::enabled_ant1 (C++ *member*), 512
- wifi_ant_config_t::rx_ant_default (C++ *member*), 511
- wifi_ant_config_t::rx_ant_mode (C++ *member*), 511
- wifi_ant_config_t::tx_ant_mode (C++ *member*), 511
- wifi_ant_gpio_config_t (C++ *class*), 511
- wifi_ant_gpio_config_t::gpio_cfg (C++ *member*), 511
- wifi_ant_gpio_t (C++ *class*), 511
- wifi_ant_gpio_t::gpio_num (C++ *member*), 511
- wifi_ant_gpio_t::gpio_select (C++ *member*), 511
- WIFI_ANT_MAX (C++ *enumerator*), 518
- WIFI_ANT_MODE_ANT0 (C++ *enumerator*), 519
- WIFI_ANT_MODE_ANT1 (C++ *enumerator*), 519
- WIFI_ANT_MODE_AUTO (C++ *enumerator*), 519
- WIFI_ANT_MODE_MAX (C++ *enumerator*), 519
- wifi_ant_mode_t (C++ *enum*), 519
- wifi_ant_t (C++ *enum*), 517
- wifi_ap_config_t (C++ *class*), 506
- wifi_ap_config_t::authmode (C++ *member*), 507
- wifi_ap_config_t::beacon_interval (C++ *member*), 507
- wifi_ap_config_t::channel (C++ *member*), 507
- wifi_ap_config_t::max_connection (C++ *member*), 507
- wifi_ap_config_t::password (C++ *member*), 507
- wifi_ap_config_t::ssid (C++ *member*), 507
- wifi_ap_config_t::ssid_hidden (C++ *member*), 507
- wifi_ap_config_t::ssid_len (C++ *member*), 507
- wifi_ap_record_t (C++ *class*), 505
- wifi_ap_record_t::ant (C++ *member*), 506
- wifi_ap_record_t::authmode (C++ *member*), 506
- wifi_ap_record_t::bssid (C++ *member*), 505
- wifi_ap_record_t::country (C++ *member*), 506
- wifi_ap_record_t::group_cipher (C++ *member*), 506
- wifi_ap_record_t::pairwise_cipher (C++ *member*), 506
- wifi_ap_record_t::phy_11b (C++ *member*), 506
- wifi_ap_record_t::phy_11g (C++ *member*), 506
- wifi_ap_record_t::phy_11n (C++ *member*), 506
- wifi_ap_record_t::phy_lr (C++ *member*), 506
- wifi_ap_record_t::primary (C++ *member*), 505
- wifi_ap_record_t::reserved (C++ *member*), 506
- wifi_ap_record_t::rssi (C++ *member*), 506
- wifi_ap_record_t::second (C++ *member*), 506
- wifi_ap_record_t::ssid (C++ *member*), 505
- wifi_ap_record_t::wps (C++ *member*), 506
- WIFI_AUTH_MAX (C++ *enumerator*), 516
- wifi_auth_mode_t (C++ *enum*), 515
- WIFI_AUTH_OPEN (C++ *enumerator*), 515
- WIFI_AUTH_WEP (C++ *enumerator*), 516
- WIFI_AUTH_WPA2_ENTERPRISE (C++ *enumerator*), 516
- WIFI_AUTH_WPA2_PSK (C++ *enumerator*), 516
- WIFI_AUTH_WPA2_WPA3_PSK (C++ *enumerator*), 516
- WIFI_AUTH_WPA3_PSK (C++ *enumerator*), 516
- WIFI_AUTH_WPA_PSK (C++ *enumerator*), 516
- WIFI_AUTH_WPA_WPA2_PSK (C++ *enumerator*), 516
- wifi_bandwidth_t (C++ *enum*), 518
- WIFI_BW_HT20 (C++ *enumerator*), 518
- WIFI_BW_HT40 (C++ *enumerator*), 518
- WIFI_CIPHER_TYPE_AES_CM40 (C++ *enumerator*), 517
- WIFI_CIPHER_TYPE_AES_CM128 (C++ *enumerator*), 517
- WIFI_CIPHER_TYPE_CCMP (C++ *enumerator*), 517
- WIFI_CIPHER_TYPE_NONE (C++ *enumerator*), 517
- wifi_cipher_type_t (C++ *enum*), 517
- WIFI_CIPHER_TYPE_TKIP (C++ *enumerator*), 517
- WIFI_CIPHER_TYPE_TKIP_CCMP (C++ *enumerator*), 517
- WIFI_CIPHER_TYPE_UNKNOWN (C++ *enumerator*), 517
- WIFI_CIPHER_TYPE_WEP104 (C++ *enumerator*), 517
- WIFI_CIPHER_TYPE_WEP40 (C++ *enumerator*), 517

- wifi_config_t (C++ union), 504
- wifi_config_t::ap (C++ member), 504
- wifi_config_t::sta (C++ member), 504
- WIFI_CONNECT_AP_BY_SECURITY (C++ enumerator), 518
- WIFI_CONNECT_AP_BY_SIGNAL (C++ enumerator), 518
- WIFI_COUNTRY_POLICY_AUTO (C++ enumerator), 515
- WIFI_COUNTRY_POLICY_MANUAL (C++ enumerator), 515
- wifi_country_policy_t (C++ enum), 515
- wifi_country_t (C++ class), 504
- wifi_country_t::cc (C++ member), 504
- wifi_country_t::max_tx_power (C++ member), 504
- wifi_country_t::nchan (C++ member), 504
- wifi_country_t::policy (C++ member), 504
- wifi_country_t::schan (C++ member), 504
- wifi_csi_cb_t (C++ type), 504
- wifi_csi_config_t (C++ class), 510
- wifi_csi_config_t::channel_filter_en (C++ member), 510
- wifi_csi_config_t::htlftf_en (C++ member), 510
- wifi_csi_config_t::lltftf_en (C++ member), 510
- wifi_csi_config_t::lftf_merge_en (C++ member), 510
- wifi_csi_config_t::manu_scale (C++ member), 511
- wifi_csi_config_t::shift (C++ member), 511
- wifi_csi_config_t::stbc_htlftf2_en (C++ member), 510
- WIFI_CSI_ENABLED (C macro), 503
- wifi_csi_info_t (C++ class), 511
- wifi_csi_info_t::buf (C++ member), 511
- wifi_csi_info_t::first_word_invalid (C++ member), 511
- wifi_csi_info_t::len (C++ member), 511
- wifi_csi_info_t::mac (C++ member), 511
- wifi_csi_info_t::rx_ctrl (C++ member), 511
- WIFI_DEFAULT_RX_BA_WIN (C macro), 503
- WIFI_DEFAULT_TX_BA_WIN (C macro), 503
- WIFI_DYNAMIC_TX_BUFFER_NUM (C macro), 503
- wifi_err_reason_t (C++ enum), 516
- wifi_event_ap_probe_req_rx_t (C++ class), 513
- wifi_event_ap_probe_req_rx_t::mac (C++ member), 514
- wifi_event_ap_probe_req_rx_t::rssi (C++ member), 514
- WIFI_EVENT_AP_PROBEREQRCVED (C++ enumerator), 521
- WIFI_EVENT_AP_STACONNECTED (C++ enumerator), 521
- wifi_event_ap_staconnected_t (C++ class), 513
- wifi_event_ap_staconnected_t::aid (C++ member), 513
- wifi_event_ap_staconnected_t::mac (C++ member), 513
- WIFI_EVENT_AP_STADISCONNECTED (C++ enumerator), 521
- wifi_event_ap_stadisconnected_t (C++ class), 513
- wifi_event_ap_stadisconnected_t::aid (C++ member), 513
- wifi_event_ap_stadisconnected_t::mac (C++ member), 513
- WIFI_EVENT_AP_START (C++ enumerator), 521
- WIFI_EVENT_AP_STOP (C++ enumerator), 521
- WIFI_EVENT_MASK_ALL (C macro), 515
- WIFI_EVENT_MASK_AP_PROBEREQRCVED (C macro), 515
- WIFI_EVENT_MASK_NONE (C macro), 515
- WIFI_EVENT_MAX (C++ enumerator), 522
- WIFI_EVENT_SCAN_DONE (C++ enumerator), 521
- WIFI_EVENT_STA_AUTHMODE_CHANGE (C++ enumerator), 521
- wifi_event_sta_authmode_change_t (C++ class), 512
- wifi_event_sta_authmode_change_t::new_mode (C++ member), 513
- wifi_event_sta_authmode_change_t::old_mode (C++ member), 513
- WIFI_EVENT_STA_BEACON_TIMEOUT (C++ enumerator), 521
- WIFI_EVENT_STA_CONNECTED (C++ enumerator), 521
- wifi_event_sta_connected_t (C++ class), 512
- wifi_event_sta_connected_t::authmode (C++ member), 512
- wifi_event_sta_connected_t::bssid (C++ member), 512
- wifi_event_sta_connected_t::channel (C++ member), 512
- wifi_event_sta_connected_t::ssid (C++ member), 512
- wifi_event_sta_connected_t::ssid_len (C++ member), 512
- WIFI_EVENT_STA_DISCONNECTED (C++ enumerator), 521
- wifi_event_sta_disconnected_t (C++ class), 512
- wifi_event_sta_disconnected_t::bssid (C++ member), 512
- wifi_event_sta_disconnected_t::reason (C++ member), 512
- wifi_event_sta_disconnected_t::ssid (C++ member), 512
- wifi_event_sta_disconnected_t::ssid_len (C++ member), 512

- WIFI_PHY_RATE_MCS0_SGI (C++ *enumerator*), 520
- WIFI_PHY_RATE_MCS1_LGI (C++ *enumerator*), 520
- WIFI_PHY_RATE_MCS1_SGI (C++ *enumerator*), 520
- WIFI_PHY_RATE_MCS2_LGI (C++ *enumerator*), 520
- WIFI_PHY_RATE_MCS2_SGI (C++ *enumerator*), 520
- WIFI_PHY_RATE_MCS3_LGI (C++ *enumerator*), 520
- WIFI_PHY_RATE_MCS3_SGI (C++ *enumerator*), 520
- WIFI_PHY_RATE_MCS4_LGI (C++ *enumerator*), 520
- WIFI_PHY_RATE_MCS4_SGI (C++ *enumerator*), 520
- WIFI_PHY_RATE_MCS5_LGI (C++ *enumerator*), 520
- WIFI_PHY_RATE_MCS5_SGI (C++ *enumerator*), 520
- WIFI_PHY_RATE_MCS6_LGI (C++ *enumerator*), 520
- WIFI_PHY_RATE_MCS6_SGI (C++ *enumerator*), 520
- WIFI_PHY_RATE_MCS7_LGI (C++ *enumerator*), 520
- WIFI_PHY_RATE_MCS7_SGI (C++ *enumerator*), 521
- wifi_phy_rate_t (C++ *enum*), 519
- WIFI_PKT_CTRL (C++ *enumerator*), 519
- WIFI_PKT_DATA (C++ *enumerator*), 519
- WIFI_PKT_MGMT (C++ *enumerator*), 519
- WIFI_PKT_MISC (C++ *enumerator*), 519
- wifi_pkt_rx_ctrl_t (C++ *class*), 509
- wifi_pkt_rx_ctrl_t::__pad0__ (C++ *member*), 509
- wifi_pkt_rx_ctrl_t::__pad1__ (C++ *member*), 509
- wifi_pkt_rx_ctrl_t::__pad2__ (C++ *member*), 509
- wifi_pkt_rx_ctrl_t::__pad3__ (C++ *member*), 509
- wifi_pkt_rx_ctrl_t::__pad4__ (C++ *member*), 509
- wifi_pkt_rx_ctrl_t::__pad5__ (C++ *member*), 510
- wifi_pkt_rx_ctrl_t::__pad6__ (C++ *member*), 510
- wifi_pkt_rx_ctrl_t::__pad7__ (C++ *member*), 510
- wifi_pkt_rx_ctrl_t::aggregation (C++ *member*), 509
- wifi_pkt_rx_ctrl_t::ampdu_cnt (C++ *member*), 509
- wifi_pkt_rx_ctrl_t::ant (C++ *member*), 510
- wifi_pkt_rx_ctrl_t::channel (C++ *member*), 509
- wifi_pkt_rx_ctrl_t::cwb (C++ *member*), 509
- wifi_pkt_rx_ctrl_t::fec_coding (C++ *member*), 509
- wifi_pkt_rx_ctrl_t::mcs (C++ *member*), 509
- wifi_pkt_rx_ctrl_t::noise_floor (C++ *member*), 509
- wifi_pkt_rx_ctrl_t::not_sounding (C++ *member*), 509
- wifi_pkt_rx_ctrl_t::rate (C++ *member*), 509
- wifi_pkt_rx_ctrl_t::rssi (C++ *member*), 509
- wifi_pkt_rx_ctrl_t::rx_state (C++ *member*), 510
- wifi_pkt_rx_ctrl_t::secondary_channel (C++ *member*), 509
- wifi_pkt_rx_ctrl_t::sgi (C++ *member*), 509
- wifi_pkt_rx_ctrl_t::sig_len (C++ *member*), 510
- wifi_pkt_rx_ctrl_t::sig_mode (C++ *member*), 509
- wifi_pkt_rx_ctrl_t::smoothing (C++ *member*), 509
- wifi_pkt_rx_ctrl_t::stbc (C++ *member*), 509
- wifi_pkt_rx_ctrl_t::timestamp (C++ *member*), 510
- wifi_pmf_config_t (C++ *class*), 506
- wifi_pmf_config_t::capable (C++ *member*), 506
- wifi_pmf_config_t::required (C++ *member*), 506
- WIFI_PROMIS_CTRL_FILTER_MASK_ACK (C *macro*), 515
- WIFI_PROMIS_CTRL_FILTER_MASK_ALL (C *macro*), 514
- WIFI_PROMIS_CTRL_FILTER_MASK_BA (C *macro*), 514
- WIFI_PROMIS_CTRL_FILTER_MASK_BAR (C *macro*), 514
- WIFI_PROMIS_CTRL_FILTER_MASK_CFEND (C *macro*), 515
- WIFI_PROMIS_CTRL_FILTER_MASK_CFENDACK (C *macro*), 515
- WIFI_PROMIS_CTRL_FILTER_MASK_CTS (C *macro*), 514
- WIFI_PROMIS_CTRL_FILTER_MASK_PSPOLL (C *macro*), 514
- WIFI_PROMIS_CTRL_FILTER_MASK_RTS (C *macro*), 514
- WIFI_PROMIS_CTRL_FILTER_MASK_WRAPPER (C *macro*), 514
- WIFI_PROMIS_FILTER_MASK_ALL (C *macro*), 514
- WIFI_PROMIS_FILTER_MASK_CTRL (C *macro*), 514
- WIFI_PROMIS_FILTER_MASK_DATA (C *macro*),

- 514
- WIFI_PROMIS_FILTER_MASK_DATA_AMPDU (C macro), 514
- WIFI_PROMIS_FILTER_MASK_DATA_MPDU (C macro), 514
- WIFI_PROMIS_FILTER_MASK_FCSFAIL (C macro), 514
- WIFI_PROMIS_FILTER_MASK_MGMT (C macro), 514
- WIFI_PROMIS_FILTER_MASK_MISC (C macro), 514
- wifi_promiscuous_cb_t (C++ type), 503
- wifi_promiscuous_filter_t (C++ class), 510
- wifi_promiscuous_filter_t::filter_mask (C++ member), 510
- wifi_promiscuous_pkt_t (C++ class), 510
- wifi_promiscuous_pkt_t::payload (C++ member), 510
- wifi_promiscuous_pkt_t::rx_ctrl (C++ member), 510
- wifi_promiscuous_pkt_type_t (C++ enum), 519
- WIFI_PROTOCOL_11B (C macro), 514
- WIFI_PROTOCOL_11G (C macro), 514
- WIFI_PROTOCOL_11N (C macro), 514
- WIFI_PROTOCOL_LR (C macro), 514
- wifi_prov_cb_event_t (C++ enum), 915
- wifi_prov_cb_func_t (C++ type), 915
- wifi_prov_config_data_handler (C++ function), 917
- wifi_prov_config_get_data_t (C++ class), 917
- wifi_prov_config_get_data_t::conn_info (C++ member), 918
- wifi_prov_config_get_data_t::fail_reason (C++ member), 918
- wifi_prov_config_get_data_t::wifi_state (C++ member), 918
- wifi_prov_config_handlers (C++ class), 918
- wifi_prov_config_handlers::apply_config_handler (C++ member), 918
- wifi_prov_config_handlers::ctx (C++ member), 918
- wifi_prov_config_handlers::get_status_handler (C++ member), 918
- wifi_prov_config_handlers::set_config_handler (C++ member), 918
- wifi_prov_config_handlers_t (C++ type), 918
- wifi_prov_config_set_data_t (C++ class), 918
- wifi_prov_config_set_data_t::bssid (C++ member), 918
- wifi_prov_config_set_data_t::channel (C++ member), 918
- wifi_prov_config_set_data_t::password (C++ member), 918
- wifi_prov_config_set_data_t::ssid (C++ member), 918
- (C++ member), 918
- WIFI_PROV_CRED_FAIL (C++ enumerator), 915
- WIFI_PROV_CRED_RECV (C++ enumerator), 915
- WIFI_PROV_CRED_SUCCESS (C++ enumerator), 916
- wifi_prov_ctx_t (C++ type), 918
- WIFI_PROV_DEINIT (C++ enumerator), 916
- WIFI_PROV_END (C++ enumerator), 916
- WIFI_PROV_EVENT_HANDLER_NONE (C macro), 915
- wifi_prov_event_handler_t (C++ class), 914
- wifi_prov_event_handler_t::event_cb (C++ member), 914
- wifi_prov_event_handler_t::user_data (C++ member), 914
- WIFI_PROV_INIT (C++ enumerator), 915
- wifi_prov_mgr_config_t (C++ class), 914
- wifi_prov_mgr_config_t::app_event_handler (C++ member), 915
- wifi_prov_mgr_config_t::scheme (C++ member), 915
- wifi_prov_mgr_config_t::scheme_event_handler (C++ member), 915
- wifi_prov_mgr_configure_sta (C++ function), 913
- wifi_prov_mgr_deinit (C++ function), 910
- wifi_prov_mgr_disable_auto_stop (C++ function), 911
- wifi_prov_mgr_endpoint_create (C++ function), 912
- wifi_prov_mgr_endpoint_register (C++ function), 912
- wifi_prov_mgr_endpoint_unregister (C++ function), 913
- wifi_prov_mgr_event_handler (C++ function), 913
- wifi_prov_mgr_get_wifi_disconnect_reason (C++ function), 913
- wifi_prov_mgr_get_wifi_state (C++ function), 913
- wifi_prov_mgr_init (C++ function), 910
- wifi_prov_mgr_is_provisioned (C++ function), 910
- wifi_prov_mgr_set_app_info (C++ function), 912
- wifi_prov_mgr_start_provisioning (C++ function), 910
- wifi_prov_mgr_stop_provisioning (C++ function), 911
- wifi_prov_mgr_wait (C++ function), 911
- wifi_prov_scheme (C++ class), 914
- wifi_prov_scheme::delete_config (C++ member), 914
- wifi_prov_scheme::new_config (C++ member), 914
- wifi_prov_scheme::prov_start (C++ member), 914
- wifi_prov_scheme::prov_stop (C++ mem-

- ber), 914
- wifi_prov_scheme::set_config_endpoint (C++ member), 914
- wifi_prov_scheme::set_config_service (C++ member), 914
- wifi_prov_scheme::wifi_mode (C++ member), 914
- wifi_prov_scheme_ble_event_cb_free_ble (C++ function), 916
- wifi_prov_scheme_ble_event_cb_free_bt (C++ function), 916
- wifi_prov_scheme_ble_event_cb_free_bt_driver (C++ function), 916
- WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREEBLE (C macro), 916
- WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREEBT (C macro), 916
- WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREEBT_DRIVER (C macro), 916
- wifi_prov_scheme_ble_set_service_uuid (C++ function), 916
- wifi_prov_scheme_softap_set_httpd_handle (C++ function), 917
- wifi_prov_scheme_t (C++ type), 915
- wifi_prov_security (C++ enum), 916
- WIFI_PROV_SECURITY_0 (C++ enumerator), 916
- WIFI_PROV_SECURITY_1 (C++ enumerator), 916
- wifi_prov_security_t (C++ type), 915
- WIFI_PROV_STA_AP_NOT_FOUND (C++ enumerator), 919
- WIFI_PROV_STA_AUTH_ERROR (C++ enumerator), 919
- wifi_prov_sta_conn_info_t (C++ class), 917
- wifi_prov_sta_conn_info_t::auth_mode (C++ member), 917
- wifi_prov_sta_conn_info_t::bssid (C++ member), 917
- wifi_prov_sta_conn_info_t::channel (C++ member), 917
- wifi_prov_sta_conn_info_t::ip_addr (C++ member), 917
- wifi_prov_sta_conn_info_t::ssid (C++ member), 917
- WIFI_PROV_STA_CONNECTED (C++ enumerator), 919
- WIFI_PROV_STA_CONNECTING (C++ enumerator), 919
- WIFI_PROV_STA_DISCONNECTED (C++ enumerator), 919
- wifi_prov_sta_fail_reason_t (C++ enum), 919
- wifi_prov_sta_state_t (C++ enum), 919
- WIFI_PROV_START (C++ enumerator), 915
- WIFI_PS_MAX_MODEM (C++ enumerator), 518
- WIFI_PS_MIN_MODEM (C++ enumerator), 518
- WIFI_PS_NONE (C++ enumerator), 518
- wifi_ps_type_t (C++ enum), 518
- WIFI_REASON_4WAY_HANDSHAKE_TIMEOUT (C++ enumerator), 516
- WIFI_REASON_802_1X_AUTH_FAILED (C++ enumerator), 516
- WIFI_REASON_AKMP_INVALID (C++ enumerator), 516
- WIFI_REASON_AP_TSF_RESET (C++ enumerator), 517
- WIFI_REASON_ASSOC_EXPIRE (C++ enumerator), 516
- WIFI_REASON_ASSOC_FAIL (C++ enumerator), 517
- WIFI_REASON_ASSOC_LEAVE (C++ enumerator), 516
- WIFI_REASON_ASSOC_NOT_AUTHED (C++ enumerator), 516
- WIFI_REASON_ASSOC_TOOMANY (C++ enumerator), 516
- WIFI_REASON_AUTH_EXPIRE (C++ enumerator), 516
- WIFI_REASON_AUTH_FAIL (C++ enumerator), 517
- WIFI_REASON_AUTH_LEAVE (C++ enumerator), 516
- WIFI_REASON_BEACON_TIMEOUT (C++ enumerator), 517
- WIFI_REASON_CIPHER_SUITE_REJECTED (C++ enumerator), 516
- WIFI_REASON_CONNECTION_FAIL (C++ enumerator), 517
- WIFI_REASON_DISASSOC_PWRCAP_BAD (C++ enumerator), 516
- WIFI_REASON_DISASSOC_SUPCHAN_BAD (C++ enumerator), 516
- WIFI_REASON_GROUP_CIPHER_INVALID (C++ enumerator), 516
- WIFI_REASON_GROUP_KEY_UPDATE_TIMEOUT (C++ enumerator), 516
- WIFI_REASON_HANDSHAKE_TIMEOUT (C++ enumerator), 517
- WIFI_REASON_IE_IN_4WAY_DIFFERS (C++ enumerator), 516
- WIFI_REASON_IE_INVALID (C++ enumerator), 516
- WIFI_REASON_INVALID_PMKID (C++ enumerator), 516
- WIFI_REASON_INVALID_RSN_IE_CAP (C++ enumerator), 516
- WIFI_REASON_MIC_FAILURE (C++ enumerator), 516
- WIFI_REASON_NO_AP_FOUND (C++ enumerator), 517
- WIFI_REASON_NOT_ASSOCED (C++ enumerator), 516
- WIFI_REASON_NOT_AUTHED (C++ enumerator), 516
- WIFI_REASON_PAIRWISE_CIPHER_INVALID (C++ enumerator), 516
- WIFI_REASON_UNSPECIFIED (C++ enumerator), 516

- WIFI_REASON_UNSUPP_RSN_IE_VERSION
(C++ *enumerator*), 516
- wifi_scan_config_t (C++ *class*), 505
- wifi_scan_config_t::bssid (C++ *member*),
505
- wifi_scan_config_t::channel (C++ *mem-
ber*), 505
- wifi_scan_config_t::scan_time (C++
member), 505
- wifi_scan_config_t::scan_type (C++
member), 505
- wifi_scan_config_t::show_hidden (C++
member), 505
- wifi_scan_config_t::ssid (C++ *member*),
505
- wifi_scan_method_t (C++ *enum*), 518
- wifi_scan_threshold_t (C++ *class*), 506
- wifi_scan_threshold_t::authmode (C++
member), 506
- wifi_scan_threshold_t::rssi (C++ *mem-
ber*), 506
- wifi_scan_time_t (C++ *class*), 505
- wifi_scan_time_t::active (C++ *member*),
505
- wifi_scan_time_t::passive (C++ *member*),
505
- WIFI_SCAN_TYPE_ACTIVE (C++ *enumerator*), 517
- WIFI_SCAN_TYPE_PASSIVE (C++ *enumerator*),
517
- wifi_scan_type_t (C++ *enum*), 517
- WIFI_SECOND_CHAN_ABOVE (C++ *enumerator*),
517
- WIFI_SECOND_CHAN_BELOW (C++ *enumerator*),
517
- WIFI_SECOND_CHAN_NONE (C++ *enumerator*), 517
- wifi_second_chan_t (C++ *enum*), 517
- WIFI_SOFTAP_BEACON_MAX_LEN (C *macro*), 503
- wifi_sort_method_t (C++ *enum*), 518
- wifi_sta_config_t (C++ *class*), 507
- wifi_sta_config_t::bssid (C++ *member*),
507
- wifi_sta_config_t::bssid_set (C++ *mem-
ber*), 507
- wifi_sta_config_t::channel (C++ *member*),
507
- wifi_sta_config_t::listen_interval
(C++ *member*), 507
- wifi_sta_config_t::password (C++ *mem-
ber*), 507
- wifi_sta_config_t::pmf_cfg (C++ *member*),
508
- wifi_sta_config_t::scan_method (C++
member), 507
- wifi_sta_config_t::sort_method (C++
member), 507
- wifi_sta_config_t::ssid (C++ *member*), 507
- wifi_sta_config_t::threshold (C++ *mem-
ber*), 507
- wifi_sta_info_t (C++ *class*), 508
- wifi_sta_info_t::mac (C++ *member*), 508
- wifi_sta_info_t::phy_11b (C++ *member*),
508
- wifi_sta_info_t::phy_11g (C++ *member*),
508
- wifi_sta_info_t::phy_11n (C++ *member*),
508
- wifi_sta_info_t::phy_lr (C++ *member*), 508
- wifi_sta_info_t::reserved (C++ *member*),
508
- wifi_sta_info_t::rssi (C++ *member*), 508
- wifi_sta_list_t (C++ *class*), 508
- wifi_sta_list_t::num (C++ *member*), 508
- wifi_sta_list_t::sta (C++ *member*), 508
- WIFI_STATIC_TX_BUFFER_NUM (C *macro*), 503
- WIFI_STATIS_ALL (C *macro*), 515
- WIFI_STATIS_BUFFER (C *macro*), 515
- WIFI_STATIS_DIAG (C *macro*), 515
- WIFI_STATIS_HW (C *macro*), 515
- WIFI_STATIS_RXTX (C *macro*), 515
- WIFI_STORAGE_FLASH (C++ *enumerator*), 518
- WIFI_STORAGE_RAM (C++ *enumerator*), 518
- wifi_storage_t (C++ *enum*), 518
- WIFI_TASK_CORE_ID (C *macro*), 503
- WIFI_VENDOR_IE_ELEMENT_ID (C *macro*), 514
- wifi_vendor_ie_id_t (C++ *enum*), 518
- wifi_vendor_ie_type_t (C++ *enum*), 518
- WIFI_VND_IE_ID_0 (C++ *enumerator*), 519
- WIFI_VND_IE_ID_1 (C++ *enumerator*), 519
- WIFI_VND_IE_TYPE_ASSOC_REQ (C++ *enumera-
tor*), 518
- WIFI_VND_IE_TYPE_ASSOC_RESP (C++ *enumera-
tor*), 518
- WIFI_VND_IE_TYPE_BEACON (C++ *enumerator*),
518
- WIFI_VND_IE_TYPE_PROBE_REQ (C++ *enumera-
tor*), 518
- WIFI_VND_IE_TYPE_PROBE_RESP (C++ *enumera-
tor*), 518
- wl_erase_range (C++ *function*), 996
- wl_handle_t (C++ *type*), 997
- WL_INVALID_HANDLE (C *macro*), 997
- wl_mount (C++ *function*), 996
- wl_read (C++ *function*), 997
- wl_sector_size (C++ *function*), 997
- wl_size (C++ *function*), 997
- wl_unmount (C++ *function*), 996
- wl_write (C++ *function*), 997
- WPS_FAIL_REASON_MAX (C++ *enumerator*), 522
- WPS_FAIL_REASON_NORMAL (C++ *enumerator*),
522
- WPS_FAIL_REASON_RECV_M2D (C++ *enumerator*),
522
- ## X
- xEventGroupClearBits (C++ *function*), 1110

- [xEventGroupClearBitsFromISR \(C macro\), 1113](#)
[xEventGroupCreate \(C++ function\), 1107](#)
[xEventGroupCreateStatic \(C++ function\), 1108](#)
[xEventGroupGetBits \(C macro\), 1115](#)
[xEventGroupGetBitsFromISR \(C++ function\), 1113](#)
[xEventGroupSetBits \(C++ function\), 1110](#)
[xEventGroupSetBitsFromISR \(C macro\), 1114](#)
[xEventGroupSync \(C++ function\), 1111](#)
[xEventGroupWaitBits \(C++ function\), 1108](#)
[xQueueAddToSet \(C++ function\), 1066](#)
[xQueueCreate \(C macro\), 1067](#)
[xQueueCreateSet \(C++ function\), 1065](#)
[xQueueCreateStatic \(C macro\), 1067](#)
[xQueueGenericCreate \(C++ function\), 1065](#)
[xQueueGenericCreateStatic \(C++ function\), 1065](#)
[xQueueGenericReceive \(C++ function\), 1062](#)
[xQueueGenericSend \(C++ function\), 1060](#)
[xQueueGenericSendFromISR \(C++ function\), 1059](#)
[xQueueGiveFromISR \(C++ function\), 1060](#)
[xQueueIsQueueEmptyFromISR \(C++ function\), 1060](#)
[xQueueIsQueueFullFromISR \(C++ function\), 1060](#)
[xQueueOverwrite \(C macro\), 1071](#)
[xQueueOverwriteFromISR \(C macro\), 1076](#)
[xQueuePeek \(C macro\), 1072](#)
[xQueuePeekFromISR \(C++ function\), 1061](#)
[xQueueReceive \(C macro\), 1074](#)
[xQueueReceiveFromISR \(C++ function\), 1063](#)
[xQueueRemoveFromSet \(C++ function\), 1066](#)
[xQueueReset \(C macro\), 1078](#)
[xQueueSelectFromSet \(C++ function\), 1066](#)
[xQueueSelectFromSetFromISR \(C++ function\), 1066](#)
[xQueueSend \(C macro\), 1070](#)
[xQueueSendFromISR \(C macro\), 1077](#)
[xQueueSendToBack \(C macro\), 1069](#)
[xQueueSendToBackFromISR \(C macro\), 1075](#)
[xQueueSendToFront \(C macro\), 1068](#)
[xQueueSendToFrontFromISR \(C macro\), 1075](#)
[xRingbufferAddToQueueSetRead \(C++ function\), 1128](#)
[xRingbufferCanRead \(C++ function\), 1128](#)
[xRingbufferCreate \(C++ function\), 1124](#)
[xRingbufferCreateNoSplit \(C++ function\), 1124](#)
[xRingbufferCreateStatic \(C++ function\), 1124](#)
[xRingbufferGetCurFreeSize \(C++ function\), 1128](#)
[xRingbufferGetMaxItemSize \(C++ function\), 1127](#)
[xRingbufferPrintInfo \(C++ function\), 1129](#)
[xRingbufferReceive \(C++ function\), 1125](#)
[xRingbufferReceiveFromISR \(C++ function\), 1126](#)
[xRingbufferReceiveSplit \(C++ function\), 1126](#)
[xRingbufferReceiveSplitFromISR \(C++ function\), 1126](#)
[xRingbufferReceiveUpTo \(C++ function\), 1126](#)
[xRingbufferReceiveUpToFromISR \(C++ function\), 1127](#)
[xRingbufferRemoveFromQueueSetRead \(C++ function\), 1128](#)
[xRingbufferSend \(C++ function\), 1124](#)
[xRingbufferSendAcquire \(C++ function\), 1125](#)
[xRingbufferSendComplete \(C++ function\), 1125](#)
[xRingbufferSendFromISR \(C++ function\), 1124](#)
[xSemaphoreCreateBinary \(C macro\), 1079](#)
[xSemaphoreCreateBinaryStatic \(C macro\), 1079](#)
[xSemaphoreCreateCounting \(C macro\), 1089](#)
[xSemaphoreCreateCountingStatic \(C macro\), 1090](#)
[xSemaphoreCreateMutex \(C macro\), 1086](#)
[xSemaphoreCreateMutexStatic \(C macro\), 1086](#)
[xSemaphoreCreateRecursiveMutex \(C macro\), 1087](#)
[xSemaphoreCreateRecursiveMutexStatic \(C macro\), 1088](#)
[xSemaphoreGetMutexHolder \(C macro\), 1091](#)
[xSemaphoreGive \(C macro\), 1082](#)
[xSemaphoreGiveFromISR \(C macro\), 1084](#)
[xSemaphoreGiveRecursive \(C macro\), 1083](#)
[xSemaphoreTake \(C macro\), 1080](#)
[xSemaphoreTakeFromISR \(C macro\), 1085](#)
[xSemaphoreTakeRecursive \(C macro\), 1081](#)
[xSTATIC_RINGBUFFER \(C++ class\), 1129](#)
[xTASK_SNAPSHOT \(C++ class\), 1056](#)
[xTASK_SNAPSHOT::pxEndOfStack \(C++ member\), 1056](#)
[xTASK_SNAPSHOT::pxTCB \(C++ member\), 1056](#)
[xTASK_SNAPSHOT::pxTopOfStack \(C++ member\), 1056](#)
[xTASK_STATUS \(C++ class\), 1056](#)
[xTASK_STATUS::eCurrentState \(C++ member\), 1056](#)
[xTASK_STATUS::pcTaskName \(C++ member\), 1056](#)
[xTASK_STATUS::pxStackBase \(C++ member\), 1056](#)
[xTASK_STATUS::ulRunTimeCounter \(C++ member\), 1056](#)
[xTASK_STATUS::usStackHighWaterMark \(C++ member\), 1056](#)
[xTASK_STATUS::uxBasePriority \(C++ member\), 1056](#)
[xTASK_STATUS::uxCurrentPriority \(C++](#)

- member*), 1056
- xTASK_STATUS::xCoreID (C++ *member*), 1056
- xTASK_STATUS::xHandle (C++ *member*), 1056
- xTASK_STATUS::xTaskNumber (C++ *member*), 1056
- xTaskCallApplicationTaskHook (C++ *function*), 1049
- xTaskCreate (C++ *function*), 1037
- xTaskCreatePinnedToCore (C++ *function*), 1037
- xTaskCreateStatic (C++ *function*), 1039
- xTaskCreateStaticPinnedToCore (C++ *function*), 1039
- xTaskGetApplicationTaskTag (C++ *function*), 1048
- xTaskGetIdleTaskHandle (C++ *function*), 1049
- xTaskGetIdleTaskHandleForCPU (C++ *function*), 1049
- xTaskGetTickCount (C++ *function*), 1047
- xTaskGetTickCountFromISR (C++ *function*), 1047
- xTaskNotify (C++ *function*), 1052
- xTaskNotifyFromISR (C++ *function*), 1052
- xTaskNotifyGive (C *macro*), 1057
- xTaskNotifyWait (C++ *function*), 1053
- xTaskResumeAll (C++ *function*), 1046
- xTaskResumeFromISR (C++ *function*), 1045
- xtensa_perfmon_config (C++ *class*), 1189
- xtensa_perfmon_config::call_function (C++ *member*), 1189
- xtensa_perfmon_config::call_params (C++ *member*), 1189
- xtensa_perfmon_config::callback (C++ *member*), 1189
- xtensa_perfmon_config::callback_params (C++ *member*), 1189
- xtensa_perfmon_config::counters_size (C++ *member*), 1190
- xtensa_perfmon_config::max_deviation (C++ *member*), 1189
- xtensa_perfmon_config::repeat_count (C++ *member*), 1189
- xtensa_perfmon_config::select_mask (C++ *member*), 1190
- xtensa_perfmon_config::tracelevel (C++ *member*), 1190
- xtensa_perfmon_config_t (C++ *type*), 1190
- xtensa_perfmon_dump (C++ *function*), 1189
- xtensa_perfmon_exec (C++ *function*), 1189
- xtensa_perfmon_init (C++ *function*), 1188
- xtensa_perfmon_overflow (C++ *function*), 1188
- xtensa_perfmon_reset (C++ *function*), 1188
- xtensa_perfmon_start (C++ *function*), 1188
- xtensa_perfmon_stop (C++ *function*), 1188
- xtensa_perfmon_value (C++ *function*), 1188
- xtensa_perfmon_view_cb (C++ *function*), 1189
- xTimerChangePeriod (C *macro*), 1099
- xTimerChangePeriodFromISR (C *macro*), 1105
- xTimerCreate (C++ *function*), 1091
- xTimerCreateStatic (C++ *function*), 1093
- xTimerDelete (C *macro*), 1100
- xTimerGetExpiryTime (C++ *function*), 1096
- xTimerGetPeriod (C++ *function*), 1096
- xTimerGetTimerDaemonTaskHandle (C++ *function*), 1096
- xTimerIsTimerActive (C++ *function*), 1096
- xTimerPendFunctionCall (C++ *function*), 1098
- xTimerPendFunctionCallFromISR (C++ *function*), 1096
- xTimerReset (C *macro*), 1101
- xTimerResetFromISR (C *macro*), 1106
- xTimerStart (C *macro*), 1098
- xTimerStartFromISR (C *macro*), 1103
- xTimerStop (C *macro*), 1099
- xTimerStopFromISR (C *macro*), 1104